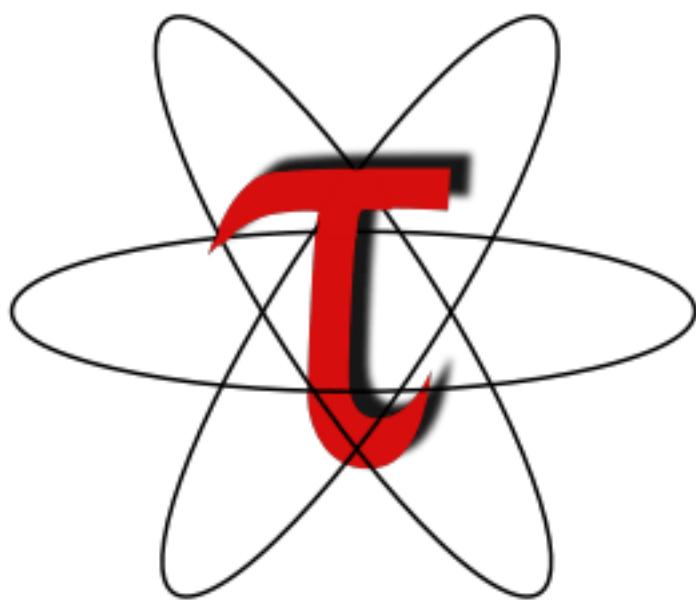


# **TAU Reference Guide**



---

## **TAU Reference Guide**

Updated December 13, 2011, for use with version 2.21.1 or greater.

Copyright © 1997-2011 Department of Computer and Information Science, University of Oregon Advanced Computing Laboratory, LANL, NM Research Centre Julich, ZAM, Germany

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of University of Oregon (UO) Research Centre Julich, (ZAM) and Los Alamos National Laboratory (LANL) not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. The University of Oregon, ZAM and LANL make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

UO, ZAM AND LANL DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE UNIVERSITY OF OREGON, ZAM OR LANL BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

TAU can be found on the web at: <http://www.cs.uoregon.edu/research/tau>

---

---

---

# Table of Contents

1. Installation .....	1
1.1. Installing TAU .....	1
1.1.1. Know what options you will need .....	1
1.1.2. Common configuration options .....	2
1.1.3. Configuring with external packages .....	3
1.1.4. More configuration options .....	4
1.1.5. tau_setup .....	8
1.1.6. installtau script .....	8
1.1.7. upgradetau .....	9
1.1.8. tau_validate .....	9
1.2. Platforms Supported .....	9
1.3. Software Requirements .....	11
2. TAU Instrumentation Options .....	12
2.1. Selective Instrumentation Options .....	12
2.2. Running an application using DynInstAPI .....	13
2.3. Rewriting Binaries using Maqau .....	13
2.4. Profiling each call to a function .....	14
2.5. Profiling with Hardware counters .....	14
2.6. Using Hardware Performance Counters .....	20
2.7. Profiling with PerfLib .....	20
2.8. Running a Python application with TAU .....	21
2.9. pprof .....	22
2.10. Running a JAVA application with TAU .....	22
2.11. Using a tau.conf File .....	23
2.12. Using Score-P with TAU .....	23
2.13. Using UPC with TAU .....	23
3. Tracing .....	25
3.1. How to configure tracing .....	25
4. TAU Memory Profiling Tutorial .....	28
4.1. TAU's memory API options .....	28
4.2. Evaluating Memory Utilization .....	28
4.2.1. TAU_TRACK_MEMORY .....	28
4.2.2. TAU_TRACK_MEMORY_HERE .....	29
4.2.3. -PROFILEMEMORY .....	29
4.3. Evaluating Memory Headroom .....	29
4.3.1. TAU_TRACK_MEMORY_HEADROOM() .....	29
4.3.2. TAU_TRACK_MEMORY_HEADROOM_HERE() .....	30
4.3.3. -PROFILEHEADROOM .....	30
4.4. DetectingMemoryLeaks .....	31
4.5. Memory Tracking In Fortran .....	32
5. Eclipse Tau Java System .....	34
5.1. Installation .....	34
5.2. Instrumentation .....	34
5.3. Uninstrumentation .....	35
5.4. Running Java with TAU .....	36
5.5. Options .....	37
6. Eclipse PTP / CDT plug-in System .....	38
6.1. Installation .....	38
6.2. Creating a Tau Launch Configuration .....	38
6.3. Selective Instrumentation .....	40
6.4. Launching a Program and Collecting Data .....	41
7. Tools .....	42
tau_compiler.sh .....	43

---

vtf2profile .....	46
tau2vtf .....	47
tau2profile .....	48
tau2elg .....	49
tau2slog2 .....	50
tau2otf .....	51
perf2tau .....	52
tau_merge .....	53
tau_treemerge.pl .....	55
tau_convert .....	56
tau_reduce .....	58
tau_ompcheck .....	60
tau_poe .....	61
tau_validate .....	62
tauex .....	63
tau_exec .....	65
tau_timecorrect .....	67
tau_throttle.sh .....	68
tau_portal.py .....	69
perfdmf_configure .....	70
perfdmf_createapp .....	71
perfdmf_createexp .....	72
perfdmf_loadtrial .....	73
perfexplorer .....	75
perfexplorer_configure .....	76
taucc .....	77
taucxx .....	78
tauf90 .....	79
paraprof .....	80
pprof .....	81
tau_instrumentor .....	82
vtfconverter .....	83
tau_setup .....	84
tau_wrap .....	85
tau_gen_wrapper .....	86
tau_pin .....	87
tau_java .....	88
tau_cupti_avail .....	89
tau_run .....	90
tau_rewrite .....	91
3. Windows .....	9
3.1. TAU on Windows .....	9
3.1.1. Installation .....	9
3.1.2. Instrumenting an application with Visual Studio C/C++ .....	9
3.1.3. Using MINGW with TAU .....	9
I. TAU Instrumentation API .....	11
TAU_START .....	14
TAU_STOP .....	15
TAU_PROFILE .....	16
TAU_DYNAMIC_PROFILE .....	17
TAU_PROFILE_CREATE_DYNAMIC .....	18
TAU_CREATE_DYNAMIC_AUTO .....	20
TAU_PROFILE_DYNAMIC_ITER .....	21
TAU_PHASE_DYNAMIC_ITER .....	22
TAU_PROFILE_TIMER .....	23
TAU_PROFILE_START .....	25
TAU_PROFILE_STOP .....	26
TAU_STATIC_TIMER_START .....	27

TAU_STATIC_TIMER_STOP .....	28
TAU_DYNAMIC_TIMER_START .....	29
TAU_DYNAMIC_TIMER_STOP .....	30
TAU_PROFILE_TIMER_DYNAMIC .....	31
TAU_PROFILE_DECLARE_TIMER .....	33
TAU_PROFILE_CREATE_TIMER .....	34
TAU_GLOBAL_TIMER .....	35
TAU_GLOBAL_TIMER_EXTERNAL .....	36
TAU_GLOBAL_TIMER_START .....	37
TAU_GLOBAL_TIMER_STOP .....	38
TAU_PHASE .....	39
TAU_DYNAMIC_PHASE .....	40
TAU_PHASE_CREATE_DYNAMIC .....	42
TAU_PHASE_CREATE_STATIC .....	44
TAU_PHASE_START .....	46
TAU_PHASE_STOP .....	47
TAU_DYNAMIC_PHASE_START .....	48
TAU_DYNAMIC_PHASE_STOP .....	49
TAU_STATIC_PHASE_START .....	50
TAU_STATIC_PHASE_STOP .....	51
TAU_GLOBAL_PHASE .....	52
TAU_GLOBAL_PHASE_EXTERNAL .....	53
TAU_GLOBAL_PHASE_START .....	54
TAU_GLOBAL_PHASE_STOP .....	55
TAU_PROFILE_EXIT .....	56
TAU_REGISTER_THREAD .....	57
TAU_PROFILE_GET_NODE .....	58
TAU_PROFILE_GET_CONTEXT .....	59
TAU_PROFILE_SET_THREAD .....	60
TAU_PROFILE_GET_THREAD .....	62
TAU_PROFILE_SET_NODE .....	63
TAU_PROFILE_SET_CONTEXT .....	65
TAU_REGISTER_FORK .....	67
TAU_REGISTER_EVENT .....	68
TAU_PROFILER_REGISTER_EVENT .....	69
TAU_EVENT .....	70
TAU_EVENT_THREAD .....	71
TAU_REGISTER_CONTEXT_EVENT .....	72
TAU_CONTEXT_EVENT .....	74
TAU_ENABLE_CONTEXT_EVENT .....	76
TAU_DISABLE_CONTEXT_EVENT .....	77
TAU_EVENT_SET_NAME .....	78
TAU_EVENT_DISABLE_MAX .....	79
TAU_EVENT_DISABLE_MEAN .....	80
TAU_EVENT_DISABLE_MIN .....	81
TAU_EVENT_DISABLE_STDDEV .....	82
TAU_REPORT_STATISTICS .....	83
TAU_REPORT_THREAD_STATISTICS .....	84
TAU_ENABLE_INSTRUMENTATION .....	85
TAU_DISABLE_INSTRUMENTATION .....	87
TAU_ENABLE_GROUP .....	89
TAU_DISABLE_GROUP .....	90
TAU_PROFILE_TIMER_SET_GROUP .....	91
TAU_PROFILE_TIMER_SET_GROUP_NAME .....	92
TAU_PROFILE_TIMER_SET_NAME .....	93
TAU_PROFILE_TIMER_SET_TYPE .....	94
TAU_PROFILE_SET_GROUP_NAME .....	95
TAU_INIT .....	96

TAU_PROFILE_INIT .....	97
TAU_GET_PROFILE_GROUP .....	98
TAU_ENABLE_GROUP_NAME .....	99
TAU_DISABLE_GROUP_NAME .....	101
TAU_ENABLE_ALL_GROUPS .....	102
TAU_DISABLE_ALL_GROUPS .....	103
TAU_GET_EVENT_NAMES .....	104
TAU_GET_EVENT_VALS .....	105
TAU_GET_COUNTER_NAMES .....	107
TAU_GET_FUNC_NAMES .....	108
TAU_GET_FUNC_VALS .....	109
TAU_ENABLE_TRACKING_MEMORY .....	111
TAU_DISABLE_TRACKING_MEMORY .....	112
TAU_TRACK_MEMORY .....	113
TAU_TRACK_MEMORY_HERE .....	114
TAU_ENABLE_TRACKING_MEMORY_HEADROOM .....	116
TAU_DISABLE_TRACKING_MEMORY_HEADROOM .....	117
TAU_TRACK_MEMORY_HEADROOM .....	118
TAU_TRACK_MEMORY_HEADROOM_HERE .....	120
TAU_SET_INTERRUPT_INTERVAL .....	121
CT .....	122
TAU_TYPE_STRING .....	123
TAU_DB_DUMP .....	125
TAU_DB_MERGED_DUMP .....	126
TAU_DB_DUMP_INCR .....	127
TAU_DB_DUMP_PREFIX .....	128
TAU_DB_DUMP_PREFIX_TASK .....	129
TAU_DB_PURGE .....	130
TAU_DUMP_FUNC_NAMES .....	131
TAU_DUMP_FUNC_VALS .....	132
TAU_DUMP_FUNC_VALS_INCR .....	133
TAU_PROFILE_STMT .....	134
TAU_PROFILE_CALLSTACK .....	135
TAU_TRACE_RECVMSG .....	136
TAU_TRACE_SENDMSG .....	138
TAU_PROFILE_PARAM1L .....	140
TAU_PROFILE_SNAPSHOT .....	141
TAU_PROFILE_SNAPSHOT_1L .....	142
TAU_PROFILER_CREATE .....	143
TAU_CREATE_TASK .....	144
TAU_PROFILER_START .....	145
TAU_PROFILER_START_TASK .....	146
TAU_PROFILER_STOP .....	147
TAU_PROFILER_STOP_TASK .....	148
TAU_PROFILER_GET_CALLS .....	149
TAU_PROFILER_GET_CALLS_TASK .....	150
TAU_PROFILER_GET_CHILD_CALLS .....	151
TAU_PROFILER_GET_CHILD_CALLS_TASK .....	152
TAU_PROFILER_GET_INCLUSIVE_VALUES .....	153
TAU_PROFILER_GET_INCLUSIVE_VALUES_TASK .....	154
TAU_PROFILER_GET_EXCLUSIVE_VALUES .....	155
TAU_PROFILER_GET_EXCLUSIVE_VALUES_TASK .....	156
TAU_PROFILER_GET_COUNTER_INFO .....	157
TAU_PROFILER_GET_COUNTER_INFO_TASK .....	158
TAU_QUERY_DECLARE_EVENT .....	159
TAU_QUERY_GET_CURRENT_EVENT .....	160
TAU_QUERY_GET_EVENT_NAME .....	161
TAU_QUERY_GET_PARENT_EVENT .....	162

II. TAU Mapping API .....	163
TAU_MAPPING .....	164
TAU_MAPPING_CREATE .....	165
TAU_MAPPING_LINK .....	167
TAU_MAPPING_OBJECT .....	169
TAU_MAPPING_PROFILE .....	170
TAU_MAPPING_PROFILE_START .....	171
TAU_MAPPING_PROFILE_STOP .....	172
TAU_MAPPING_PROFILE_TIMER .....	173
A. Environment Variables .....	174

---

## List of Figures

5.1. TAUJava Options Screen .....	34
5.2. TAUJava Project Instrumentation .....	34
5.3. TAUJava Running .....	36
6.1. TAU Setup .....	38
6.2. TAU Launch Configuration .....	39
6.3. Optional User Defined Events .....	40
6.4. Adding User Defined Events .....	40

---

## List of Tables

2.1. Events measured by setting the environment variable PAPI_EVENT in TAU .....	15
2.2. Events measured by setting the environment variable PCL_EVENT in TAU .....	18
7.1. Selection Attributes .....	58
A.1. TAU Environment Variables .....	174

---

# Chapter 1. Installation

TAU (Tuning and Analysis Utilities) is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C++, C, Java and Python. The model that TAU uses to profile parallel, multi-threaded programs maintains performance data for each thread, context, and node in use by an application. The profiling instrumentation needed to implement the model captures data for functions, methods, basic blocks, and statement execution at these levels. All C++ language features are supported in the TAU profiling instrumentation including templates and namespaces, which is available through an API at the library or application level. The API also provides selection of profiling groups for organizing and controlling instrumentation. The instrumentation can be inserted in the source code using an automatic instrumentor tool based on the Program Database Toolkit (PDT), dynamically using DyninstAPI, at runtime in the Java virtual machine, or manually using the instrumentation API. TAU's profile visualization tool, paraprof, provides graphical displays of all the performance analysis results, in aggregate and single node/context/thread forms. The user can quickly identify sources of performance bottlenecks in the application using the graphical interface. In addition, TAU can generate event traces that can be displayed with the Vampir or Paraver trace visualization tools. This chapter discusses installation of the TAU portable profiling package.

## 1.1. Installing TAU

### 1.1.1. Know what options you will need

Each TAU configuration results in a single `Makefile`. These `Makefiles` denote the configuration that produced it and is used by the user to select the TAU libraries/scripts associated with its configuration. (These makefiles are named after the configuration options, ie: TAU configured with MPI, PDT, PGI compilers and the '`-nocomm`' option is named: `Makefile.tau-nocomm-mpi-pdt-pgi`). On most machines several configuration of TAU will need to be built in order to take full advantage of the many features of TAU. This section should help you decide on the smallest set of configuration you will need to build.

The options used to configure TAU can be grouped into two categories:

- External packages: TAU will use these when instrumenting or measuring an application. *Configuring with these options does not force the user to use these packages*, ie: configuring with PDT does not force the user to use source code based instrumentation (they can use compiler based instrumentation instead). Similarly configuring with PAPI does not forces the user to select any PAPI counters when profiling.



#### Note

The only exception is configuring with the epilog (scalasca) tracing package. This will replace the TAU tracer with the epilog one, a single configuration cannot use both tracers.

For this reason it is recommend that you *configure with every external packages that the user might be interested in using*, letting them choose which packages to enable when they go to instrument or measure their application.

- Compiler and MPI options: these control the behavior of TAU when it compiles the instrumented application. TAU provides compiler wrapper scripts, these options control which compiler TAU will wrap, *These options are determinative: select only options that are compatible*. For example, when configuring with MPI use a version of MPI compatible with the compiler you select.

Since multiple compiler/MPI libraries cannot be specified for a single configuration, *each set of compiler/MPI libraries that you want to use with TAU need to be configured separately*.



## Note

Configurations with different compilers are given separate `Makefiles` automatically, however configurations with different MPI implementations are not. Use the `-tag=` option to distinguish between different MPIs, ie: `-tag=mvapich` or `-tag=openmpi`.

The `configure` shell script attempts to guess correct values for various system-dependent variables used during compilation (compilers and system architecture), other options need to be specified on the command line.

The following are the most important command-line options available to configure:

## 1.1.2. Common configuration options

### 1.1.2.1. Select compiler

TAU defaults to using any compilers found in the environment. To use a specific compiler use these options:

- `-c++=<C++ Compiler>`  
Specifies the name of the C++ compiler. Supported C++ compilers include KCC (from KAI/Intel), CC (SGI, Sun), g++ (from GNU), FCC (from Fujitsu), xlC (from IBM), guidec++ (from KAI/Intel), cxx (Tru64) and aCC (from HP), c++ (from Apple), icpc and ecpc (from Intel) and pgCC (from PGI).
- `-cc=<C Compiler>`  
Specifies the name of the C compiler. Supported C compilers include cc, gcc (from GNU), pgcc (from PGI), fcc (from Fujitsu), xlc (from IBM), and KCC (from KAI/Intel),icc and ecc (from Intel).
- `-fortran=<Fortran Compiler>`  
Specifies the name of the Fortran90 compiler. Valid options are: gnu, sgi, ibm, ibm64, intel, cray, pgi, abssoft, fujitsu, sun, kai, nec, hitachi, compaq, nagware, g95 and hp.

### 1.1.2.2. Using MPI

To profile a program that use MPI configure TAU with these options:

- `-mpi`

With this option, TAU will try to guess the location of the MPI libraries if the `mpirun` command is in your path. This does not always work in which case use these more detailed options:

- `-mpiinc=<dir>`

Specifies the directory where MPI header files reside (such as `mpi.h` and `mpif.h`). This option also

generates the TAU MPI wrapper library that instruments MPI routines using the MPI Profiling Interface. See the examples/NPB2.3/config/make.def file for its usage with Fortran and MPI programs. MPI [<http://www-unix.mcs.anl.gov/mpi/>]

- `-mpilib=<dir>`

Specifies the directory where MPI library files reside. This option should be used in conjunction with the `-mpiinc=<dir>` option to generate the TAU MPI wrapper library.

- `-mpilibrary=<lib>`

Specifies the use of a different MPI library. By default, TAU uses `-lmpi` or `-lmpich` as the MPI library. This option allows the user to specify another library. e.g., `-mpilibrary=-lmpi_r` for specifying a thread-safe MPI library.

### 1.1.2.3. OpenMP

To profile programs using openmp use `-openmp` and either OPARI option:

- `-openmp`

Specifies OpenMP as the threads package to be used. Open MPI [<http://www.open-mpi.org/>]

- `-opari`

The use of Opari2 source-to-source instrumentor in conjunction with TAU exposes OpenMP events for instrumentation. See examples/opari directory. OPARI [<http://www.vi-hps.org/projects/score-p/>]

- `-opari1`

Use this option for the use of the original Opari. Only use this option if `-opari` fails. OPARI [<http://www.fz-juelich.de/zam/kojak/opari/>]

### 1.1.3. Configuring with external packages

- `-pdt=<directory>`

Specifies the location of the installed PDT (Program Database Toolkit) root directory. PDT is used to build tau\_instrumentor, a C++, C and F90 instrumentation program that automatically inserts TAU annotations in the source code PDT [<http://www.cs.uoregon.edu/research/pdt>]. If PDT is configured with a subdirectory option (`-compdir=<opt>`) then TAU can be configured with the same option.

- `-papi=<directory>`

Specifies the location of the installed PAPI (Performance Data Standard and API) root directory. PCL provides a common interface to access hardware performance counters and timers on modern microprocessors. Most modern CPUs provide on-chip hardware performance counters that can record several events such as the number of instructions issued, floating point operations performed, the number of primary and secondary data and instruction cache misses. To measure floating point instructions, set the environment variable TAU\_METRICS to PAPI\_FP\_INS (for example). This option (by default) specifies the use of hardware performance counters for profiling (instead of time). PAPI [<http://icl.cs.utk.edu/papi/>]

- **-epilog=<dir>**

Specifies the directory where the EPILOG tracing package EPILOG [<http://www.fz-juelich.de/zam/kojak/epilog/>] is installed. This option should be used in conjunction with the -TRACE option to generate binary EPILOG traces (instead of binary TAU traces). EPILOG traces can then be used with other tools such as EXPERT. EPILOG comes with its own implementation of the MPI wrapper library and the POMP library used with Opari. Using option overrides TAU's libraries for MPI, and OpenMP.

- **-otf=<directory>**

Specifies the location of the OTF trace libraries generation package. TAU's binary traces can be converted to the OTF format using tau2otf, a tool that links with the OTF library.

- **-vtf=<directory>**

Specifies the location of the VTF3 trace generation package. TAU's binary traces can be converted to the VTF3 format using tau2vtf, a tool that links with the VTF3 library. The VTF3 format is read by Intel trace analyzer, formerly known as vampir, a commercial trace visualization tool developed by TU Dresden, Germany.

## 1.1.4. More configuration options

- **-PROFILEPHASE**

This option generates phase based profiles. It requires special instrumentation to mark phases in an application (I/O, computation, etc.). Phases can be static or dynamic (different phases for each loop iteration, for instance). See examples/phase/README for further information.

- **-prefix=<directory>**

Specifies the destination directory where the header, library and binary files are copied. By default, these are copied to subdirectories <arch>/bin and <arch>/lib in the TAU root directory.

- **-arch=<architecture>**

Specifies the architecture. If the user does not specify this option, configure determines the architecture. For IBM BGL, the user should specify bgl as the architecture. For SGI, the user can specify either of sgi32, sgin32 or sgi64 for 32, n32 or 64 bit compilation modes respectively. The files are installed in the <architecture>/bin and <architecture>/lib directories. Cray options are xt3, craycnl or crayxmt.

- **-pdt\_c++=<C++ Compiler>**

Specifies a different C++ compiler for PDT (tau\_instrumentor). This is typically used when the library is compiled with a C++ compiler (specified with -c++) and the tau\_instrumentor is compiled with a different <pdt\_c++> compiler. For e.g.,

```
-c++=pgCC -cc=pgCC -pdt_c++=KCC -openmp ...
```

uses PGI's OpenMP compilers for TAU's library and KCC for tau\_instrumentor.

```
-arch=bgl -pdt=/usr/pdtoolkit-3.4 -pdt_c++=xlC -mpi
```

uses PDT, MPI for IBM BG/L and specifies the use of the front-end xlC compiler for building

tau\_instrumentor.

- **-tag=<Unique Name>**

Specifies a tag in the name of the stub Makefile and TAU makefiles to uniquely identify the installation. This is useful when more than one MPI library may be used with different versions of compilers. e.g.,

```
% configure -c++=icpc -cc=icc -tag=intel71-vmi  \
             -mpiinc=/vml2/mpich/include
```

- **-scalasca=<dir>**

Specifies the directory where the SCALASCA [<http://www.sclasca.org>] package is installed.

- **-pthread**

Specifies pthread as the thread package to be used. In the default mode, no thread package is used.

- **-opari\_region**

Report performance data for only OpenMP regions and not constructs. By default, both regions and constructs are profiled with Opari.

- **-opari\_construct**

Report performance data for only OpenMP constructs and not Regions. By default, both regions and constructs are profiled with Opari.

- **-pdtarch=<architecture>**

Specifies the architecture used to build pdt, default the tau architecture.

- **-papithreads**

Same as papi, except uses threads to highlight how hardware performance counters may be used in a multi-threaded application. When it is used with PAPI, TAU should be configured with **-papi=<dir>** **-pthread** autoinstrument Shows the use of Program Database Toolkit (PDT) for automating the insertion of TAU macros in the source code. It requires configuring TAU with the **-pdt=<dir>** option. The Makefile is modified to illustrate the use of a source to source translator (tau\_instrumentor).

- **-jdk=<directory>**

Specifies the location of the installed Java root directory. TAU can profile or trace Java applications without any modifications to the source code, byte-code or the Java virtual machine. See README.JAVA on instructions on using TAU with Java 2 applications. Also the reference guide has more information on the new tau\_java tool. This option should only be used for configuring TAU to use JVMTI for profiling and tracing of Java applications. It should not be used for configuring paraprof, which uses Java from the user's path.

- **-dyninst=<dir>**

Specifies the directory where the DynInst dynamic instrumentation package is installed. Using DynInst, a user can invoke tau\_run to instrument an executable program at runtime or prior to execution by rewriting it. DyninstAPI [<http://www.dyninst.org/>] PARA-DYN [<http://www.paradyn.org/>].

- **-vampirtrace=<directory>**

Specifies the location of the Vampir Trace package. With this option TAU will generate traces in Open Trace Format (OTF). For more information, see Technische Universität Dresden [ <http://www.tu-dresden.de/zih/vampirtrace>]

- **-scorep=<directory>**

Specify location of Score-P package. Set the environment variable SCOREP\_PROFILING\_FORMAT to TAU\_SNAPSHOT so that Score-P will output Tau Snapshot profiles.

- **-shmeminc=<dir>**

Specifies the directory where shmem.h resides and specifies the use of the TAU SHMEM interface.

- **-shmemlib=<dir>**

Specifies the directory where libsma.a resides and specifies the use of the TAU SHMEM interface.

- **-shmemlibrary=<lib>**

By default, TAU uses -lsma as the shmem/pshmem library. This option allows the user to specify a different shmem library.

- **-nocomm**

Allows the user to turn off tracking of messages (synchronous/asynchronous) in TAU's MPI wrapper interposition library. Entry and exit events for MPI routines are still tracked. Affects both profiling and tracing.

- **-cuda=<dir>**

Specifies the location of the top level CUDA SDK

- **-opencl=<dir>**

Specifies the location of the OpenCL package

- **-armci=<dir>**

Specifies the location of the ARMCI directory

- **-epiloglib=<dir>**

Specifies the directory of where the Epilog library is to be found. Ex: if directory structure is: /usr/local/epilog/fe/lib/ let the install options be: -epilog=/usr/local/epilog -epiloglib=/usr/local/epilog/fe/lib.

- **-epilogbin=<dir>**

Specifies the directory of where the Epilog binaries are to be found.

- **-epiloginc=<dir>**

Specifies the directory of where the epilog's included sources headers are to be found.

- **-MPITRACE**

Specifies the tracing option and generates event traces for MPI calls and routines that are ancestors of MPI calls in the callstack. This option is useful for generating traces that are converted to the EPI-

LOG trace format. KOJAK's Expert automatic diagnosis tool needs traces with events that call MPI routines. Do not use this option with the -TRACE option.

- `-pythoninc=<dir>`

Specifies the location of the Python include directory. This is the directory where Python.h header file is located. This option enables python bindings to be generated. The user should set the environment variable PYTHONPATH to <TAUROOT>/<ARCH>/lib/bindings-<options> to use a specific version of the TAU Python bindings. By importing package pytau, a user can manually instrument the source code and use the TAU API. On the other hand, by importing tau and using tau.run(<func>), TAU can automatically generate instrumentation. See examples/python directory for further information.

- `-pythonlib=<dir>`

Specifies the location of the Python lib directory. This is the directory where \*.py and \*.pyc files (and config directory) are located. This option is mandatory for IBM when Python bindings are used. For other systems, this option may not be specified (but -pythoninc=<dir> needs to be specified).

- `-PROFILEMEMORY`

Specifies tracking heap memory utilization for each instrumented function. When any function entry takes place, a sample of the heap memory used is taken. This data is stored as user-defined event data in profiles/traces.

- `-PROFILECOMMUNICATORS`

This option generates MPI information partitioned by communicators. TAU lists upto 8 ranks in each communicator in the listing.

- `-PROFILEHEADROOM`

Specifies tracking memory available in the heap (as opposed to memory utilization tracking in - PROFILEMEMORY). When any function entry takes place, a sample of the memory available (headroom to grow) is taken. This data is stored as user-defined event data in profiles/traces. Please refer to the examples/headroom/README file for a full explanation of these headroom options and the C++/C/F90 API for evaluating the headroom.

- `-COMPENSATE`

Specifies online compensation of performance perturbation. When this option is used, TAU computes its overhead and subtracts it from the profiles. It can be only used when profiling is chosen. This option works with MULTIPLECOUNTERS as well, but while it is relevant for removing perturbation with wallclock time, it cannot accurately account for perturbation with hardware performance counts (e.g., L1 Data cache misses). See TAU Publication [Europar04] for further information on this option.

- `-PROFILECOUNTERS`

Specifies use of hardware performance counters for profiling under IRIX using the SGI R10000 perfex counter access interface. The use of this option is deprecated in favor of the -pcl=<dir> and -papi=<dir> options described above.

- `-noex`

Specifies that no exceptions be used while compiling the library. This is relevant for C++.

- `-useropt=<options-list>`

Specifies additional user options such as -g or -I. For multiple options, the options list should be enclosed in a single quote. For example

```
% ./configure -useropt='-g -I/usr/local/stl'
```

- `-mrnet=<mrnet source root>`

Base location of the MRnet package.

- `-mrnetlib=<mrnet libraries>`

Path to the MRnet libraries.



### Note

On some cluster systems the MRnet libraries need to be available to the runtime system (ie. on the lustre filesystem.)

- `-scorep=<scorep subsystem>`

Path to the Score-P measurement system. Set the environment variable SCOREP\_PROFILING\_FORMAT to TAU\_SNAPSHOT so that Score-P will output Tau Snapshot profiles.

- `-help`

Lists all the available configure options and quits.

## 1.1.5. tau\_setup

`tau_setup` is a GUI interface to the `configure` and `installtau` tools.

## 1.1.6. installtau script

To install multiple (typical) configurations of TAU at a site, you may use the script `installtau'. It takes options similar to those described above. It invokes `./configure <opts>; make clean install;` to create multiple libraries that may be requested by the users at a site. The `installtau` script accepts the following options:

```
% installtau -help

TAU Configuration Utility
*****
Usage: installtau [OPTIONS]
  where [OPTIONS] are:
-arch=<arch>
-fortran=<compiler>
-cc=<compiler>
-c++=<compiler>
-useropt=<options>
-pdt=<pdt dir>
-pdtcompdir=<compdir>
-pdt_c++=<C++ Compiler>
```

```
-papi=<papidir>
-vtf=<vtkdir>
-otf=<otkdir>
-dyninst=<dyninstdir>
-mpi
-mpiinc=<mpiincdir>
-mpilib=<mpilibdir>
-mpilibrary=<mpilibrary>
-perfinc=<dir>
-perflib=<dir>
-perflibrary=<library>
-mpi
-tag=<unique name>
-opari=<oparidir>
-epilog=<epilogdir>
-epiloginc=<absolute path to epilog include dir> (<epilog>/include default)
-epilogbin=<absolute path to epilog bin dir> (<epilog>/bin default)
-epiloglib=<absolute path to epilog lib dir> (<epilog>/lib default)
-prefix=<dir>
-exec-prefix=<dir>
-j=<num processes for parallel make> (just -j for full parallel)

*****
```

These options are similar to the options used by the configure script.

## 1.1.7. upgradetau

This script is provided to rebuild all TAU configurations previously built in a different TAU source directory. Give this command the location of a previous version of tau followed by any additional configurations and it will rebuild tau with these same options.

## 1.1.8. tau\_validate

This script will attempt to validate a tau installation. Its only argument is TAU's architecture directory. These are some options:

- -v Verbose output
- --html Output results in HTML
- --build Only build
- --run Only run

Here is a simple example:

```
bash : ./tau_validate --html x86_64 &> results.html
tcsh : ./tau_validate --html x86_64 >& results.html
```

## 1.2. Platforms Supported

TAU has been tested on the following platforms:

- 1. SGI

On IRIX 6.x based systems, including Indy, Power Challenge, Onyx, Onyx2 and Origin 200, 2000, 3000 Series, CC 7.2+, KAI [<http://www.kai.com>] KCC and g++ [<http://www.gnu.org>] compilers are supported. On SGI Altix systems, Intel, and GNU compilers are supported.

- 2. LINUX Clusters

On Linux based Intel x86 PC clusters, KAI/Intel's KCC, g++, egcs (GNU), pgCC (PGI) [<http://www.pgroup.com>], FCC (Fujitsu) [<http://www.fujitsu.com>] and icpc/ecpc Intel [<http://www.intel.com>] compilers have been tested. TAU also runs under IA-64, Opteron, PowerPC, Alpha, Apple PowerMac, Sparc and other processors running Linux.

- 3. Sun Solaris

Sun compilers (CC, F90), KAI KCC, KAP/Pro and GNU g++ work with TAU.

- 4. IBM

On IBM SP2 and AIX systems, KAI KCC, KAP/Pro, IBM xlC, xlc, xlf90 and g++ compilers work with TAU. On IBM BG/L, IBM xlC, blrts\_xlC, blrts\_xlf90, blrts\_xlc, and gnu compilers work with TAU. On IBM pSeries Linux, xlC, xlc, xlf90 and gnu compilers work with TAU.

- 5. HP HP-UX

On HP PA-RISC systems, aCC and g++ can be used.

- 6. HP Alpha Tru64

On HP Alpha Tru64 machines, cxx and g++, and Guide compilers may be used with TAU.

- 7. NEC SX series vector machines

On NEC SX-5 systems, NEC c++ may be used with TAU.

- 8. Cray X1, T3E, SV-1, XT3, RedStorm, and Cray Compute Node Linux (CNL)

On Cray T3E systems, KAI KCC and Cray CC compilers have been tested with TAU. On Cray SV-1 and X1 systems, Cray CC compilers have been tested with TAU. On Cray XT3, and RedStorm systems, PGI and GNU compilers have been tested with TAU. For those using Cray CNL you need to configure tau with the option `-arch=craycnl`

- On Hitachi machines, Hitachi KCC, g++ and Hitachi cc compilers may be used with TAU

- 10. Apple OS X

On Apple OS X machines, c++ or g++ may be used to compile TAU. Also, IBM's xlf90, xlf and Absoft Fortran 90 compilers for G4/G5 may be used with TAU.

- 11. Fujitsu PRIMEPOWER

On Fujitsu Power machines, Sun and Fujitsu compilers may be used with TAU.

- 11. Microsoft Window

On Windows, Microsoft Visual C++ 6.0 or higher and JDK 1.2+ compilers have been tested with TAU

NOTE: TAU has been tested with JDK 1.2, 1.3, 1.4.x under Solaris, SGI, IBM, Linux, and MacOS X.

## 1.3. Software Requirements

- 1. Java v 1.5

TAU's GUI ParaProf and PerfExplorer require Java v1.4 or better in your path. If Java 1.4 is the only version available, older version of ParaProf and PerfExplorer can be installed. To do so, simply run either program with Java 1.4 in your path. You will be guided through the installation process. ParaProf does not require `-jdk=<dir>` option to be specified during configuration. (This option is used for configuring TAU for analyzing Java applications.)

---

# Chapter 2. TAU Instrumentation Options

## 2.1. Selective Instrumentation Options

**Selective Instrumentation File Specification.** The selective instrumentation file has the following sections, each preceded and followed by:

BEGIN\_EXCLUDE\_LIST /  
END\_EXCLUDE\_LIST or BE-  
GIN\_INCLUDE\_LIST /  
END\_INCLUDE\_LIST

exclude/include list of routines and/or files for instrumentation. The list of routines to be excluded from instrumentation is specified, one per line, enclosed by BEGIN\_EXCLUDE\_LIST and END\_EXCLUDE\_LIST. Instead of specifying which routines should be excluded, the user can specify the list of routines that are to be instrumented using the include list, one routine name per line, enclosed by BEGIN\_INCLUDE\_LIST and END\_INCLUDE\_LIST.

BEGIN\_FILE\_EXCLUDE\_LIST /  
END\_FILE\_EXCLUDE\_LIST or  
BEGIN\_FILE\_INCLUDE\_LIST /  
END\_FILE\_INCLUDE\_LIST

Similarly, files can be included or excluded with the BE-  
GIN\_FILE\_EXCLUDE\_LIST,  
END\_FILE\_EXCLUDE\_LIST,  
BEGIN\_FILE\_INCLUDE\_LIST,  
and  
END\_FILE\_INCLUDE\_LIST lines.

BEGIN\_INSTRUMENT\_SECTION  
/ END\_INSTRUMENT\_SECTION

Manually editing the selective instrumentation file gives you more options. These tags allow you to control the type of instrumentation performed in certain portions of your application.

- Static and Dynamic timers can be set by specifying either a range of line numbers or a routine.

```
static timer name="foo_bar" file="foo.c" line=17 to line=18
dynamic timer routine="int fool(int)
```

- Static and Dynamic phases can be set by specifying either a range of line numbers or a routine. If you do not configure TAU with -PROFILEPHASE these phases will be converted to regular timers.

```
static phase routine="int foo(int)
dynamic phase name="fool_bar" file="foo.c" line=26 to line=27
```

- Loops in the source code can be profiled by specifying a routine in which all loop should be profiled, like:

```
loops file="loop_test.cpp" routine="multiply"
```

- With Memory Profiling the following events are tracked: memory allocation, memory deallocation,

and memory leaks.

```
memory file="foo.f90" routine="INIT"
```

- IO Events track the size, in bytes of read, write, and print statements.

```
io file="foo.f90" routine="RINB"
```

Both Memory and IO events are represented along with their call-stack; the length of which can be set with environment variable TAU\_CALLPATH\_DEPTH.



### Note

Due to the limitations of the some compilers (IBM xlf, PGI pgf90, GNU gfortran), the size of the memory reported for a Fortran Array is not the number of bytes but rather the number of elements.

## 2.2. Running an application using DynInstAPI

TAU also allows you to dynamically instrument your application using the DynInst package. There are a few limitation to DyInst: 1) only function level events will be captured and 2) your application must be compiled with debugging symbols (-g).

To install the DynInstAPI package, configure TAU with -dyinst= option which will point TAU to where dyninst is installed. Use the tau\_run tool to instrument your application at runtime.

The command-line options accepted by tau\_run are:

```
Usage: tau_run [-Xrun<TauLibrary> ][-v][ -o outfile ] \
[-f <instrumentation file> ] <application> [args]
```

By default, libTAU.so is loaded by tau\_run. However, the user can override this and specify another file using the -Xrun<TauLibrary>. In this case lib<TauLibrary>.so will be loaded using LD\_LIBRARY\_PATH.

To use tau\_run, TAU is configured with DyninstAPI as shown below:

```
% configure -dyninst=/usr/local/packages/dyninstAPI
% make install
% cd tau/examples/dyninst
% make install
% tau_run klargest 2500 23
% pprof; paraprof
```

## 2.3. Rewriting Binaries using Maqau

TAU also allows you to rewrite your application using the Maqau package included in PDTToolkit 3.17 or above(<http://tau.uoregon.edu/pdt.tgz>).

Install PDTToolkit 3.17+ and configure TAU with -pdt= option which will point TAU to where PDTToolkit is installed. Use the tau\_rewrite tool to instrument your application. (If TAU is not configured with PDT 3.17+, then tau\_rewrite defaults to tau\_run.)

```
% configure -dyninst=/usr/local/packages/pdtoolkit-3.17  
% make install  
% tau_rewrite -T scorep,pdt -loadlib=/tmp/libfoo.so ./a.out -o a.inst
```

## 2.4. Profiling each call to a function

By default TAU profiles the total time (inclusive/exclusive) spent on a given function. Profiling each function call for an application that calls some function hundred of thousands of times, is impractical since the profile data would grow enormously. But configuring TAU with the -PROFILEPARAM option will have TAU profile select functions each time they are called. But TAU will also group some of these function calls together according to the value of the parameter they are given. For example if a function mpisend(int i) is called 2000 times 1000 times with 512 and 1000 times with 1024 then we will receive two profile for mpisend() one we it is called with 512 and one when it is called with 1024. This reduces the overhead since we are profiling mpisend() two times not 2000 times.

## 2.5. Profiling with Hardware counters

LIST OF COUNTERS:

Set the following values for the COUNTER<1-25> environment variables.

- GET\_TIME\_OF\_DAY - For the default profiling option using gettimeofday()
- SGI\_TIMERS - For -SGITIMERS configuration option under IRIX
- CRAY\_TIMERS - For -CRAYTIMERS configuration option under Cray X1.
- LINUX\_TIMERS - For -LINUXTIMERS configuration option under Linux
- CPU\_TIME - For user+system time from getrusage() call with -CPUTIME
- P\_WALL\_CLOCK\_TIME - For PAPI's WALLCLOCK time using -PAPIWALLCLOCK
- P\_VIRTUAL\_TIME - For PAPI's process virtual time using -PAPIVIRTUAL
- TAU\_MUSE - For reading counts of Linux OS kernel level events when MAGNET/MUSE is installed and -muse configuration option is enabled. MUSE [<http://public.lanl.gov/radiant/>].TAU\_MUSE\_PACKAGE environment variable has to be set to package name (busy\_time, count, etc.)
- TAU\_MPI\_MESSAGE\_SIZE - For tracking the cumulative message size for all MPI operations by a node for each routine.



### Note

When TAU is configured with -TRACE -MULTIPLECOUNTERS and -papi=<dir> op-

tions, the COUNTER1 environment variable must be set to GET\_TIME\_OF\_DAY to allow TAU's tracing module to use a globally synchronized real-time clock for timestamping event records. When we use tracing with hardware performance counters, the counters specified in environment variables COUNTER[2-25] are accessed at routine transitions and logged in the trace file. Use tau2vtf tool to convert TAU traces to VTF3 traces that may be loaded in the Vampir trace visualization tool.

and PAPI/PCL options that can be found in Table 2.1, “Events measured by setting the environment variable PAPI\_EVENT in TAU” and Table 2.2, “Events measured by setting the environment variable PCL\_EVENT in TAU”. Example:

- PCL\_FP\_INSTR - For floating point operations using PCL (-pcl=<dir>)
- PAPI\_FP\_INS - For floating point operations using PAPI (-papi=<dir>)
- PAPI\_NATIVE\_<event> - For native papi events using PAPI (-papi=<dir>)

*NOTE:* When **-MULTIPLECOUNTERS** is used with **-TRACE** option, the tracing library uses the wall-clock time from the function specified in the COUNTER1 variable. This should typically point to wall-clock time routines (such as **GET\_TIME\_OF\_DAY** or **SGI\_TIMERS** or **LINUX\_TIMERS**).

Example:

```
% setenv COUNTER1 P_WALL_CLOCK_TIME  
% setenv COUNTER2 PAPI_L1_DCM  
% setenv COUNTER3 PAPI_FP_INS
```

will produce profile files in directories called **MULT\_P\_WALL\_CLOCK\_TIME**, **MULTI\_PAPI\_L1\_DCM**, and **MULTI\_PAPI\_FP\_INS**.

**Table 2.1. Events measured by setting the environment variable PAPI\_EVENT in TAU**

PAPI_EVENT	EVENT Measured
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_L2_DCM	Level 2 data cache misses
PAPI_L2_ICM	Level 2 instruction cache misses
PAPI_L3_DCM	Level 3 data cache misses
PAPI_L3_ICM	Level 3 instruction cache misses
PAPI_L1_TCM	Level 1 total cache misses
PAPI_L2_TCM	Level 2 total cache misses
PAPI_L3_TCM	Level 3 total cache misses
PAPI_CA_SNP	Snoops
PAPI_CA_SHR	Request for access to shared cache line (SMP)
PAPI_CA_CLN	Request for access to clean cache line (SMP)
PAPI_CA_INV	Cache Line Invalidiation (SMP)

PAPI_EVENT	EVENT Measured
PAPI_CA_ITV	Cache Line Intervention (SMP)
PAPI_L3_LDM	Level 3 load misses
PAPI_L3_STM	Level 3 store misses
PAPI_BRU_IDL	Cycles branch units are idle
PAPI_FXU_IDL	Cycles integer units are idle
PAPI_FPU_IDL	Cycles floating point units are idle
PAPI_LSU_IDL	Cycles load/store units are idle
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_TLB_TL	Total translation lookaside buffer misses
PAPI_L1_LDM	Level 1 load misses
PAPI_L1_STM	Level 1 store misses
PAPI_L2_LDM	Level 2 load misses
PAPI_L2_STM	Level 2 store misses
PAPI_BTAC_M	BTAC miss
PAPI_PRF_DM	Prefetch data instruction caused a miss
PAPI_L3_DCH	Level 3 Data Cache Hit
PAPI_TLB_SD	Translation lookaside buffer shootdowns (SMP)
PAPI_CSR_FAL	Failed store conditional instructions
PAPI_CSR_SUC	Successful store conditional instructions
PAPI_CSR_TOT	Total store conditional instructions
PAPI_MEM_SCY	Cycles Stalled Waiting for Memory Access
PAPI_MEM_RCY	Cycles Stalled Waiting for Memory Read
PAPI_MEM_WCY	Cycles Stalled Waiting for Memory Write
PAPI_STL_ICY	Cycles with No Instruction Issue
PAPI_FUL_ICY	Cycles with Maximum Instruction Issue
PAPI_STL_CCY	Cycles with No Instruction Completion
PAPI_FUL_CCY	Cycles with Maximum Instruction Completion
PAPI_HW_INT	Hardware interrupts
PAPI_BR_UCN	Unconditional branch instructions executed
PAPI_BR_CN	Conditional branch instructions executed
PAPI_BR_TKN	Conditional branch instructions taken
PAPI_BR_NTK	Conditional branch instructions not taken
PAPI_BR_MSP	Conditional branch instructions mispredicted
PAPI_BR_PRC	Conditional branch instructions correctly predicted
PAPI_FMA_INS	FMA instructions completed
PAPI_TOT_IIS	Total instructions issued
PAPI_TOT_INS	Total instructions executed
PAPI_INT_INS	Integer instructions executed
PAPI_FP_INS	Floating point instructions executed
PAPI_LD_INS	Load instructions executed

<b>PAPI_EVENT</b>	<b>EVENT Measured</b>
PAPI_SR_INS	Store instructions executed
PAPI_BR_INS	Total branch instructions executed
PAPI_VEC_INS	Vector/SIMD instructions executed
PAPI_FLOPS	Floating Point Instructions executed per second
PAPI_RES_STL	Cycles processor is stalled on resource
PAPI_FP_STAL	FP units are stalled
PAPI_TOT_CYC	Total cycles
PAPI_IPS	Instructions executed per second
PAPI_LST_INS	Total load/store instructions executed
PAPI_SYC_INS	Synchronization instructions executed
PAPI_L1_DCH	L1 D Cache Hit
PAPI_L2_DCH	L2 D Cache Hit
PAPI_L1_DCA	L1 D Cache Access
PAPI_L2_DCA	L2 D Cache Access
PAPI_L3_DCA	L3 D Cache Access
PAPI_L1_DCR	L1 D Cache Read
PAPI_L2_DCR	L2 D Cache Read
PAPI_L3_DCR	L3 D Cache Read
PAPI_L1_DCW	L1 D Cache Write
PAPI_L2_DCW	L2 D Cache Write
PAPI_L3_DCW	L3 D Cache Write
PAPI_L1_ICH	L1 instruction cache hits
PAPI_L2_ICH	L2 instruction cache hits
PAPI_L3_ICH	L3 instruction cache hits
PAPI_L1_ICA	L1 instruction cache accesses
PAPI_L2_ICA	L2 instruction cache accesses
PAPI_L3_ICA	L3 instruction cache accesses
PAPI_L1_ICR	L1 instruction cache reads
PAPI_L2_ICR	L2 instruction cache reads
PAPI_L3_ICR	L3 instruction cache reads
PAPI_L1_ICW	L1 instruction cache writes
PAPI_L2_ICW	L2 instruction cache writes
PAPI_L3_ICW	L3 instruction cache writes
PAPI_L1_TCH	L1 total cache hits
PAPI_L2_TCH	L2 total cache hits
PAPI_L3_TCH	L3 total cache hits
PAPI_L1_TCA	L1 total cache accesses
PAPI_L2_TCA	L2 total cache accesses
PAPI_L3_TCA	L3 total cache accesses
PAPI_L1_TCR	L1 total cache reads
PAPI_L2_TCR	L2 total cache reads

PAPI_EVENT	EVENT Measured
PAPI_L3_TCR	L3 total cache reads
PAPI_L1_TCW	L1 total cache writes
PAPI_L2_TCW	L2 total cache writes
PAPI_L3_TCW	L3 total cache writes
PAPI FML_INS	FM ins
PAPI_FAD_INS	FA ins
PAPI_FDV_INS	FD ins
PAPI_FSQ_INS	FSq ins
PAPI_FNV_INS	Finv ins

For example to measure the floating point operations in routines using PCL,

```
% ./configure -pcl=/usr/local/packages/pcl-1.2
% setenv PCL_EVENT PCL_FP_INSTR
% mpirun -np 8 application
```

**Table 2.2. Events measured by setting the environment variable PCL\_EVENT in TAU**

PCL_EVENT	EVENT Measured
PCL_L1CACHE_READ	L1 (Level one) cache reads
PCL_L1CACHE_WRITE	L1 cache writes
PCL_L1CACHE_READWRITE	L1 cache reads and writes
PCL_L1CACHE_HIT	L1 cache hits
PCL_L1CACHE_MISS	L1 cache misses
PCL_L1DCACHE_READ	L1 data cache reads
PCL_L1DCACHE_WRITE	L1 data cache writes
PCL_L1DCACHE_READWRITE	L1 data cache reads and writes
PCL_L1DCACHE_HIT	L1 data cache hits
PCL_L1DCACHE_MISS	L1 data cache misses
PCL_L1ICACHE_READ	L1 instruction cache reads
PCL_L1ICACHE_WRITE	L1 instruction cache writes
PCL_L1ICACHE_READWRITE	L1 instruction cache reads and writes
PCL_L1ICACHE_HIT	L1 instruction cache hits
PCL_L1ICACHE_MISS	L1 instruction cache misses
PCL_L2CACHE_READ	L2 (Level two) cache reads
PCL_L2CACHE_WRITE	L2 cache writes
PCL_L2CACHE_READWRITE	L2 cache reads and writes
PCL_L2CACHE_HIT	L2 cache hits
PCL_L2CACHE_MISS	L2 cache misses
PCL_L2DCACHE_READ	L2 data cache reads
PCL_L2DCACHE_WRITE	L2 data cache writes

PCL_EVENT	EVENT Measured
PCL_L2DCACHE_READWRITE	L2 data cache reads and writes
PCL_L2DCACHE_HIT	L2 data cache hits
PCL_L2DCACHE_MISS	L2 data cache misses
PCL_L2ICACHE_READ	L2 instruction cache reads
PCL_L2ICACHE_WRITE	L2 instruction cache writes
PCL_L2ICACHE_READWRITE	L2 instruction cache reads and writes
PCL_L2ICACHE_HIT	L2 instruction cache hits
PCL_L2ICACHE_MISS	L2 instruction cache misses
PCL_TLB_HIT	TLB (Translation Lookaside Buffer) hits
PCL_TLB_MISS	TLB misses
PCL_ITLB_HIT	Instruction TLB hits
PCL_ITLB_MISS	Instruction TLB misses
PCL_DTLB_HIT	Data TLB hits
PCL_DTLB_MISS	Data TLB misses
PCL_CYCLES	Cycles
PCL_ELAPSED_CYCLES	Cycles elapsed
PCL_INTEGER_INSTR	Integer instructions executed
PCL_FP_INSTR	Floating point (FP) instructions executed
PCL_LOAD_INSTR	Load instructions executed
PCL_STORE_INSTR	Store instructions executed
PCL_LOADSTORE_INSTR	Loads and stores executed
PCL_INSTR	Instructions executed
PCL_JUMP_SUCCESS	Successful jumps executed
PCL_JUMP_UNSUCCESS	Unsuccessful jumps executed
PCL_JUMP	Jumps executed
PCL_ATOMIC_SUCCESS	Successful atomic instructions executed
PCL_ATOMIC_UNSUCCESS	Unsuccessful atomic instructions executed
PCL_ATOMIC	Atomic instructions executed
PCL_STALL_INTEGER	Integer stalls
PCL_STALL_FP	Floating point stalls
PCL_STALL_JUMP	Jump stalls
PCL_STALL_LOAD	Load stalls
PCL_STALL_STORE	Store Stalls
PCL_STALL	Stalls
PCL_MFLOPS	Millions of floating point operations/second
PCL_IPC	Instructions executed per cycle
PCL_L1DCACHE_MISSRATE	Level 1 data cache miss rate
PCL_L2DCACHE_MISSRATE	Level 2 data cache miss rate
PCL_MEM_FP_RATIO	Ratio of memory accesses to FP operations

## 2.6. Using Hardware Performance Counters

While running the application, set the environment variable PCL\_EVENT or PAPI\_EVENT respectively, to specify which hardware performance counter TAU should use while profiling the application.



### Note

By default, only one counter is tracked at a time. To track more than one counter use –MULTIPLECOUNTERS. See ??? for more details.

To select floating point instructions for profiling using PAPI, you would:

```
% configure -papi=/usr/local/packages/papi-3.5.0  
% make clean install  
% cd examples/papi  
% setenv PAPI_EVENT PAPI_FP_INS  
% a.out
```

In addition to the following events, you can use native events (see **papi\_native**) on a given CPU by setting PAPI\_EVENT to PAPI\_NATIVE\_<event>. For example:

```
% setenv PAPI_EVENT PAPI_NATIVE_PM_BIQ_IDU_FULL_CYC  
% a.out
```

By default PAPI will profile events in all domains (users space, kernel, hypervisor, etc). You can restrict the set of domains for papi event profiling by using the TAU\_PAPI\_DOMAIN environment variable with these values (in a colon separated list, if desired): PAPI\_DOM\_USER, PAPI\_DOM\_KERNEL, PAPI\_DOM\_SUPERVISOR, and PAPI\_DOM\_OTHER like thus:

```
% setenv TAU_PAPI_DOMAIN PAPI_DOM_SUPERVISOR:PAPI_DOM_OTHER
```

## 2.7. Profiling with PerfLib

This profiling option is currently under development at LANL.

To configure TAU with PerfLib use the following arguments:

```
%> configure -perflib=[path_to_perflib lib directory]  
           -perfinc=[path_to_perflib inc directory]  
           -perflibrary=[argument send to the linker if different than default]
```

After tau is build a new Makefile will be generated with \*-perflib-\* in its name, use this Makefile when profiling applications with perflib.

After configuration and installation, toggle these three environment variables before running the application:

```
%> export PERF_PROFILE=1
```

```
%> export PERF_PROFILE_MPI=1
%> export PERF_PROFILE_MEMORY=1
%> export PERF_PROFILE_COUNTERS=1
%> export PERF_DATA_DIRECTORY=<directory>
```

We also provide a perf2tau conversion utilities to convert the remaining perflib profiles to regular tau profiles. To use perf2tau set the environment variable `perf_data_directory` to the type of the profiling to be converted (the directory where the data is store will be called something like `perf_data.[type]/`). Or you may execute perf2tau with the type as an argument:

```
%> perf2tau [type]
```

See also the man page for perf2tau, perf2tau.

## 2.8. Running a Python application with TAU

TAU can automatically instrument all Python routines when the tau python package is imported. Add `<TAUROOT>/<ARCH>/lib/bindings-<options>` to the `PYTHONPATH` environment variable in order to use the TAU module.

To execute the program, `tau.run` routine is invoked with the name of the top level Python code. For e.g.,

```
#!/usr/bin/env python

import tau
from time import sleep

def f2():
    print "Inside f2: sleeping for 2 secs..."
    sleep(2)
def f1():
    print "Inside f1, calling f2..."
    f2()

def OurMain():
    f1()

tau.run('OurMain()')
```

instruments routines `OurMain()`, `f1()` and `f2()` although there are no instrumentation calls in the routines. To use this feature, TAU must be configured with the `-pythoninc=<dir>` option (and `-pythonlib=<dir>` if running under IBM). Before running the application, the environment variable `PYTHONPATH` and `LD_LIBRARY_PATH` should be set to include the TAU library directory (where `tau.py` is stored). Manual instrumentation of Python sources is also possible using the Python API and the `pytau` package. For e.g.,

```
#!/usr/bin/env python

import pytau
from time import sleep

x = pytau.profileTimer("A Sleep for excl 5 secs")
y = pytau.profileTimer("B Sleep for excl 2 secs")
pytau.start(x)
```

```
print "Sleeping for 5 secs ..."
sleep(5)
pytau.start(y)
print "Sleeping for 2 secs ..."
sleep(2)
pytau.stop(y)
pytau.dbDump()
pytau.stop(x)
```

shows how two timers x and y are created and used. Note, multiple timers can be nested, but not overlapping. Overlapping timers are detected by TAU at runtime and flagged with a warning (as exclusive time is not defined when timers overlap).

## 2.9. pprof

pprof sorts and displays profile data generated by TAU. To view the profile, merely execute pprof in the directory where profile files are located (or set the PROFILEDIR environment variable).

```
% pprof
```

Its usage is explained below:

```
usage: pprof [-c|-b|-m|-t|-e|-i] [-r] [-s] [-n num] [-f filename] \
              [-l] [node numbers]
  -c : Sort by number of Calls
  -b : Sort by number of subRoutines called by a function
  -m : Sort by Milliseconds (exclusive time total)
  -t : Sort by Total milliseconds (inclusive time total) (DEFAULT)
  -e : Sort by Exclusive time per call (msec/call)
  -i : Sort by Inclusive time per call (total msec/call)
  -v : Sort by standard deviation (excl usec)
  -r : Reverse sorting order
  -s : print only Summary profile information
  -n num : print only first num functions
  -f filename : specify full path and filename without node ids
  -p : suppress conversion to hh:mm:ss:mmm format
  -l : List all functions and exit
  -d : Dump output format (for Racy) [node numbers] : prints only info about
       all contexts/threads of given node numbers
  node numbers : prints information about all contexts/threads
  for specified nodes
```

## 2.10. Running a JAVA application with TAU

Java applications are profiled/traced using tau\_java as shown below:

```
% cd tau/examples/java/pi
% setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:<tauroot>/<arch>/lib
% tau_java Pi
```

More information about tau\_java can be found in the Tools section of the Reference Guide.

Running the application generates profile files with names having the form pro-

file.<node>.<context>.<thread>. These files can be analyzed using pprof or paraprof.

## 2.11. Using a tau.conf File

If a tau.conf file is created, then code that uses that TAU lib will be effected by the settings in tau.conf. For example, if a directory tau-2.21/tau\_system\_defaults is created and a tau.conf file is placed in it, TAU will read that file before doing the measurements. A user of that TAU lib can choose to override the contents of that file by placing a tau.conf in their own directory. But by default, if the sysadmin chooses to create this dir, all the users of the TAU libs will be globally affected by this tau.conf.

For example, tau.conf could be:

```
% cat tau.conf
TAU_LOG_PATH=/soft/apps/tau/logs
PROFILEDIR=$TAU_LOG_DIR
TAU_PROFILE_FORMAT=merged
TAU_SUMMARY=1
TAU_IBM_BG_HWP_COUNTERS=1
TAU_TRACK_MESSAGE=1
```

Then anyone using TAU from that directory will get TAU\_IBM\_BG\_HWP\_COUNTERS=1, TAU\_TRACK\_MESSAGE=1, etc.

## 2.12. Using Score-P with TAU

TAU can be configured to use the Score-P measurement infrastructure ([www.score-p.org](http://www.score-p.org)). To use Score-P, configure TAU with `-scorep=` option to point TAU to the Score-P installation. (Please use Score-P version 1.0 beta or above.) You may then instrument and run your application with TAU in a manor of your choosing.

Set the environment variable SCOREP\_PROFILING\_FORMAT to TAU\_SNAPSHOT to produce TAU Snapshot files, which will be found in scorep\*/tau/. Also, the Score-P library must be found in LD\_LIBRARY\_PATH.

## 2.13. Using UPC with TAU

Please see examples/upc for more details.

To instrument Berkeley UPC with GASP, configure TAU with `-upcnetwork=<option>` where option is "mpi" or "udp". Then use a selective instrumentation file like the one shown below.

```
BEGIN_INSTRUMENT_SECTION
forall routine="#"
loops routine="#"
barrier routine="#"
fence routine="#"
notify routine="#"
END_INSTRUMENT_SECTION
```

Then tau\_upc.sh can be used to build the application. If "udp" is used with `-upcnetwork`, then upcrun can be used to run the application. For "mpi", mpirun or a similar mechanism can be used.

TAU can also build the DMAPP wrapper using Cray CCE compilers. When the `-optDMAPP` option is used when building the application with TAU using `TAU_OPTIONS`, DMAPP events are automatically instrumented with `tau_upc.sh`.

---

# Chapter 3. Tracing

## 3.1. How to configure tracing

TAU must be configured with the `-TRACE` option to generate event traces. This can be used in conjunction with `-PROFILE` to generate both profiles and traces. The traces are stored in a directory specified by the environment variable `TRACEDIR`, or the current directory, by default. The environment variables `TAU_TRACEFILE` may be used to specify the name of Vampir trace file. When this variable is set, trace files are automatically merged and the `tau2vtf` is invoked to convert the merged trace file to VTF3 trace format. This conversion takes place on node 0, thread 0. The intermediate trace files are deleted. To retain the trace files, the user can set the environment variable `TAU_KEEP_TRACEFILES` to true. When `TAU_TRACEFILE` is not specified, the user needs to merge and convert the traces as below. Example:

```
% ./configure -arch=sgi64 -TRACE -mpi -vtf=/usr/local/vtf3-1.34 -slog2  
% make clean; make install  
% setenv TRACEDIR /users/sameer/tracedata/experiment56  
% mpirun -np 4 matrix
```

This generates files named

```
tautrace.<node>.<context>.<thread>.trc and events.<node>.edf
```

When generating a Vampir Trace Format (otf or vtf) these environment variables maybe helpful:

- `VT_FILE_PREFIX`Prefix used for trace filenames. Default is "a".
- `VT_COMPRESSION`Write compressed trace files? Default is "yes"

Using the utility `tau_treemerge.pl`, these traces are then merged as shown below:

```
% tau_treemerge.pl
```

This generates `tau.trc` as the merged trace file and `tau.edf` as the merged event description file.

`tau_treemerge.pl` can take an optional argument (with `-n <value>`) to specify the maximum number of trace files to merge in each invocation of `tau_merge`. If we need to merge 2000 trace files and if the maximum number of open files specified by unix is 250, `tau_treemerge.pl` will incrementally merge the trace files so as not to exceed the number of open file descriptors. This is important for the IBM Blue-Gene/L machine where such restrictions are present on the front-end node.

To convert merged or per-thread traces to another trace format, the utilities, `tau2otf`, `tau_convert`, `tau2vtf`, or `tau2slog2` are used as shown below:

```
Usage: tau2otf [ -n streams ] [ -nomessage ] [ -v ] [ -z ]  
-n streams : Specifies the number of output streams (default is 1)  
-nomessage : Suppresses printing of message information in the trace
```

```
-v      : Verbose mode sends trace event descriptions to the standard output  
as they are converted  
-z      : Compressed output
```

Here is an example:

```
%> tau2otf tau.trc tau.edf out.otf
```

Converting to Vampir's VTF format:

```
% tau2vtf  
Usage: tau2vtf <TAU trace> <edf file> <out file> [-a|-fa]  
      [-nomessage] [-v]  
-a      : ASCII VTF3 file format  
-fa     : FAST ASCII VTF3 file format  
-nomessage : Suppress printing of message information in the trace  
-v      : Verbose  
Default trace format of <out file> is VTF3 binary  
e.g.,  
tau2vtf merged.trc tau.edf app.vpt.gz  
% tau2vtf matrix.trc tau.edf matrix.vpt.gz  
% vampir matrix.vpt.gz
```

To generate slog2 trace files that may be visualized using Jumpshot, we recommend using the slog2 SDK and Jumpshot bundled with TAU.

```
% configure -slog2 -TRACE ...  
% tau2slog2  
tau2slog2 converts a TAU formatted trace file to the SLOG2 format  
for Jumpshot trace visualizer  
Usage: tau2slog2 <tau_tracefile> <edf_file> -o <slog_tracefile>  
For e.g.,  
% tau2slog2 app.trc tau.edf -o app.slog2
```

To generate traces that may be visualized using Vampir, we recommend using tau2vtf over the older tau\_convert tool. tau2vtf can produce binary traces with user-defined events (hardware performance counters from PAPI etc.) while tau\_convert cannot do this. Binary traces load faster in Vampir.

```
% tau_convert  
usage: tau_convert [-alog | -SDDF | -dump | -paraver [-t] | -pv |  
                   -vampir [-longsymbolbugfix] [-compact] [-user|-class|-all]  
                   [-nocomm]] inputtrc edffile [outputtrc]  
Note: -vampir option assumes multiple threads/node  
Note: -t option used in conjunction with -paraver option assumes  
multiple threads/node
```

To view the dump of the trace in text form, use

```
% tau_convert -dump matrix.trc tau.edf
```

tau\_convert can also be used to convert traces to the Vampir [<http://www.vampir-ng.de/>] trace format. For single-threaded applications (such as the MPI application above), the -pv option is used to generate Vampir traces as follows:

```
% tau_convert -pv matrix.trc tau.edf matrix.pv  
% vampir matrix.vpt.gz &
```

To convert TAU traces to SDDF or ALOG trace formats, -SDDF and -alog options may be used. When multiple threads are used on a node (as with -jdk, -pthread or -tulipthread options during configure), the -vampir option is used to convert the traces to the vampir trace format, as shown below:

```
% tau_convert -vampir smartsapp.trc tau.edf smartsapp.pv  
% vampir smartsapp.pv &
```

To convert to the Paraver trace format, use the -paraver option for single threaded programs and -paraver -t option for multi-threaded programs.

*NOTE:* To ensure that inter-process communication events are recorded in the traces, in addition to the routine transitions, it is necessary to insert TAU\_TRACE\_SENDSMSG and TAU\_TRACE\_RECVMSG macro calls in the source code during instrumentation. This is not needed when the TAU MPI wrapper library is used.

Vampir format traces may be converted to TAU profiles using the vtf2profile tool.

```
% vtf2profile -f matrix.vpt.gz -p profiledatadir  
% vtf2profile  
Usage: vtf2profile [options]  
*****HELP*****  
* '-h' display this help text. *  
* '-c' open command line interface. *  
* '-f' used as -f <VTF File> where *  
* VTF File is the name of the trace file *  
* to be converted to TAU profiles. *  
* '-p' used as -p <path> where 'path' is the relative *  
* path to the directory where profiles are to *  
* stored. *  
* '-i' used as -i <from> <to> where 'from' and 'to' are*  
* integers to mark the desired profiling interval.*  
*****
```

---

# Chapter 4. TAU Memory Profiling Tutorial

## 4.1. TAU's memory API options

TAU can evaluate the following memory events:

1. Memory utilization options that examine how much heap memory is currently used, and
2. Memory headroom evaluation options that examine how much a program can grow (or how much headroom it has) before it runs out of free memory on the heap. During memory headroom evaluation TAU tries to call malloc with chunks that progressively increase in size, until all memory is exhausted. Then it frees those chunks, keeping track of how much memory it successfully allocated.
3. Memory leaks in C/C++ programs TAU will track malloc through the execution issuing user event when the program fails to free the allocated memory.

## 4.2. Evaluating Memory Utilization

### 4.2.1. TAU\_TRACK\_MEMORY

When TAU\_TRACK\_MEMORY is called an interrupt is generated every 10 seconds and the memory event is triggered with the current value. This interrupt interval can be changed by calling TAU\_SET\_INTERRUPT\_INTERVAL(value). The tracking of memory events in both cases can be explicitly enabled or disabled by calling the macros TAU\_ENABLE\_TRACKING\_MEMORY() or TAU\_DISABLE\_TRACKING\_MEMORY() respectively.

TAU\_TRACK\_MEMORY() can be inserted into the source code:

```
int main(int argc, char **argv)
{
    TAU_PROFILE("main()", " ", TAU_DEFAULT);
    TAU_PROFILE_SET_NODE(0);

    TAU_TRACK_MEMORY();

    sleep(12);

    int *x = new int[5*1024*1024];

    sleep(12);

    return 0;
}
```

Resulting profile data:

```
USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
-----
NumSamples      MaxValue      MinValue      MeanValue      Std. Dev.      Event Name
```

```
-----  
      2  2.049E+04      2.891  1.024E+04  1.024E+04  Memory Utilization  
      (heap, in KB)  
-----
```

## 4.2.2. TAU\_TRACK\_MEMORY\_HERE

Triggers memory tracking at a given execution point. For example:

```
int main(int argc, char **argv) {  
    TAU_PROFILE("main()", " ", TAU_DEFAULT);  
    TAU_PROFILE_SET_NODE(0);  
  
    TAU_TRACK_MEMORY_HERE();  
  
    int *x = new int[5*1024*1024];  
    TAU_TRACK_MEMORY_HERE();  
    return 0;  
}
```

Here is the resulting profile:

```
USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0  
-----  
NumSamples  MaxValue  MinValue  MeanValue  Std. Dev.  Event Name  
-----  
      2  2.049E+04      2.891  1.024E+04  1.024E+04  Memory Utilization  
      (heap, in KB)  
-----
```

## 4.2.3. -PROFILEMEMORY

Specifies tracking heap memory utilization for each instrumented function. When any function entry takes place, a sample of the heap memory used is taken. This data is stored as user-defined event data in profiles/traces.

# 4.3. Evaluating Memory Headroom

## 4.3.1. TAU\_TRACK\_MEMORY\_HEADROOM()

This call sets up a signal handler that is invoked every 10 seconds by an interrupt. Inside, it evaluates how much memory it can allocate and associates it with the callstack. The user can vary the size of the callstack by setting the environment variable TAU\_CALLSTACK\_DEPTH (default is 2). The examples/headroom/track subdirectory has an example that illustrates the use of this call. To disable tracking this headroom at runtime, the user may call: TAU\_DISABLE\_TRACKING\_MEMORY\_HEADROOM() and call TAU\_ENABLE\_TRACKING\_MEMORY\_HEADROOM() to re-enable tracking of the headroom. To set a different interrupt interval, call TAU\_SET\_INTERRUPT\_INTERVAL(value) where value (in seconds) represents the inter-interrupt interval.

A sample profile generated has:

```
USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
-----
NumSamples    MaxValue    MinValue    MeanValue    Std. Dev.    Event Name
-----
3            4067        4061        4065        2.828    Memory Headroom Left (in
                                         MB)
3            4067        4061        4065        2.828    Memory Headroom
Left (in MB) : void quicksort(int *, int, int)  => void
quicksort(int *, int, int)
-----
```

### 4.3.2. TAU\_TRACK\_MEMORY\_HEADROOM\_HERE()

Sometimes it is useful to track the memory available at a certain point in the program, rather than rely on an interrupt. `TAU_TRACK_MEMORY_HEADROOM_HERE()` allows us to examine the memory available at a particular location in the source code and associate it with the currently executing callstack. The examples/headroom/here subdirectory has an example that illustrates this usage.

```
ary = new double [1024*1024*50];
TAU_TRACK_MEMORY_HEADROOM_HERE(); /* takes a sample here! */
sleep(1);
```

A sample profile looks like this:

```
USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
-----
NumSamples    MaxValue    MinValue    MeanValue    Std. Dev.    Event Name
-----
3            3672        3672        3672        0          Memory Headroom Left (in
                                         MB)
1            3672        3672        3672        0          Memory Headroom
Left (in MB) : main() (calls f1, f2, f4)
1            3672        3672        3672        0          Memory
Headroom Left (in MB) : main() (calls f1, f2, f4) (sleeps 1 sec, calls f2, f4) => f4
1            3672        3672        3672        0          Memory Headroom Left (in MB) : main() (calls f1, f5) => f5
-----
```

### 4.3.3. -PROFILEHEADROOM

Similar to the `-PROFILEMEMORY` configuration option that takes a sample of the memory utilization at each function entry, we now have `-PROFILEHEADROOM`. In this `-PROFILEHEADROOM` option, a sample is taken at instrumented function's entry and associated with the function name. This option is meant to be used as a debugging aid due the high cost associated with executing a series of malloc calls. The cost was 106 microseconds on an IBM BG/L (700 MHz CPU). To use this option, simply configure TAU with the `-PROFILEHEADROOM` option and choose any method for instrumentation (PDT, MPI, hand instrumentation). You do not need to annotate the source code in any special way (as is required for 2a and 2b). The examples/headroom/available subdirectory has a simple example that produces the following profile when TAU is configured with the `-PROFILEHEADROOM` option.

```
USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
```

NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.	Event	Name
1	4071	4071	4071	0	f1()	(sleeps 1 sec, calls f2, f4) - Memory Headroom Available (MB)
2	3671	3671	3671	0	f2()	(sleeps 2 sec, calls f3) - Memory Headroom Available (MB)
2	3671	3671	3671	0	f3()	(sleeps 3 sec) - Memory Headroom Available (MB)
1	3671	3671	3671	0	f4()	(sleeps 4 sec, calls f2) - Memory Headroom Available (MB)
1	3671	3671	3671	0	f5()	(sleeps 5 sec) - Memory Headroom Available (MB)
1	4071	4071	4071	0	main()	(calls f1, f5) - Memory Headroom Available (MB)

## 4.4. DetectingMemoryLeaks

TAU's memory leak detection feature can be initiated by giving `tau_compiler.sh` the option `-optDetectMemoryLeaks`. For a demonstration consider this C++ program:

```
#include <stdio.h>
#include <malloc.h>

/* there is a memory leak in bar when it is invoked with 5 < value <= 15 */
int bar(int value)
{
    printf("Inside bar: %d\n", value);
    int *x;

    if (value > 5)
    {
        printf("looks like it came here from g!\n");
        x = (int *) malloc(sizeof(int) * value);
        x[2] = 2;
        /* do not free it! create a memory leak, unless the value is > 15 */
        if (value > 15) free(x);
    }
    else
    { /* value <=5 no leak */
        printf("looks like it came here from foo!\n");
        x = (int *) malloc(sizeof(int) * 45);
        x[23] = 2;
        free(x);
    }
    return 0;
}

int g(int value)
{
    printf("Inside g: %d\n", value);
    return bar(value);
}

int foo(int value)
{
    printf("Inside f: %d\n", value);
```

```
    if (value > 5) g(value);
    else bar(value);

    return 0;
}
int main(int argc, char **argv)
{
    int *x;
    int *y;
    printf ("Inside main\n");

    foo(12); /* leak */
    foo(20); /* no leak */
    foo(2); /* no leak */
    foo(13); /* leak */
}
```

Notice that bar fails to free allocated memory on input between 5 and 15 and that foo will call g that calls bar when the input to foo is greater than 5.

Now configuring TAU with `-PROFILECALLPATH` run the file by:

```
%> cd examples/memoryleakdetect/
%> make
%> ./simple
...
USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
-----
NumSamples  MaxValue  MinValue  MeanValue  Std. Dev.  Event Name
-----
      2        52        48        50          2  MEMORY LEAK! malloc size <
      1        80        80        80          0  free size <file=simple.ins
      1        80        80        80          0  free size <file=simple.ins
      1       180       180       180          0  free size <file=simple.ins
      1       180       180       180          0  free size <file=simple.ins
      3        80        48        60         14.24  malloc size <file=simple.i
      3        80        48        60         14.24  malloc size <file=simple.i
      1       180       180       180          0  malloc size <file=simple.i
      1       180       180       180          0  malloc size <file=simple.i
```

Notice that the first row show the two Memory leaks along with the callpath tracing where the unallocated memory was requested.

## 4.5. Memory Tracking In Fortran

To profile memory usage in Fortran 90 use TAU's ability to selectively instrument a program. The option `-optTauSelectFile=<file>` for `tau_compliler.sh` let you specify a selective instrumentation file which defines regions of the source code to instrument.

To begin memory profiling, state which file/routines to profile by typing:

```
BEGIN_INSTRUMENT_SECTION
memory file="memory.f90" routine="INIT"
END_INSTRUMENT_SECTION
```

Wildcard can be used to instrument multiple routines. For file names \* character can be used to specify any number of character, thus foo\* matches foobar, foo2, etc. also for file names ? can match a single character, ie. foo? matches foo2, fooZ, but not foobar. You can use # as a wildcard for routines, ie. b# matches bar, b2z etc.

Memory Profile in Fortran gives you these three metrics:

- Total size of memory for each `malloc` and `free` in the source code.
- The callpath for each occurrence of `malloc` or `free`.
- A list of all variable that were not deallocated in the source code.



## Note

Due to the limitations of the `xlf` compiler, The size of the memory reported for Fortran Array (compilated with `xlf`) is not the number of bytes but the number of elements.

Here is the profile for the `example/memoryleakdetect/f90/foof90.f90` file.

```
%> pprof
...
-----
NumSamples      MaxValue      MinValue      MeanValue      Std. Dev.      Event Name
-----
1              16             16             16             0      MEMORY LEAK! malloc size <
2              52             48             50             2      MEMORY LEAK! malloc size <
1              80             80             80             0      free size <file=foo.f90, v
1              80             80             80             0      free size <file=foo.f90, v
1              180            180            180            0      free size <file=foo.f90, v
1              180            180            180            0      free size <file=foo.f90, v
1              180            180            180            0      malloc size <file=foo.f90,
1              180            180            180            0      malloc size <file=foo.f90,
1              180            180            180            0      malloc size <file=foo.f90,
4              80             16             49             22.69     malloc size <file=foo.f90,
1              16             16             16             0      malloc size <file=foo.f90,
3              80             48             60             14.24     malloc size <file=foo.f90,
```

---

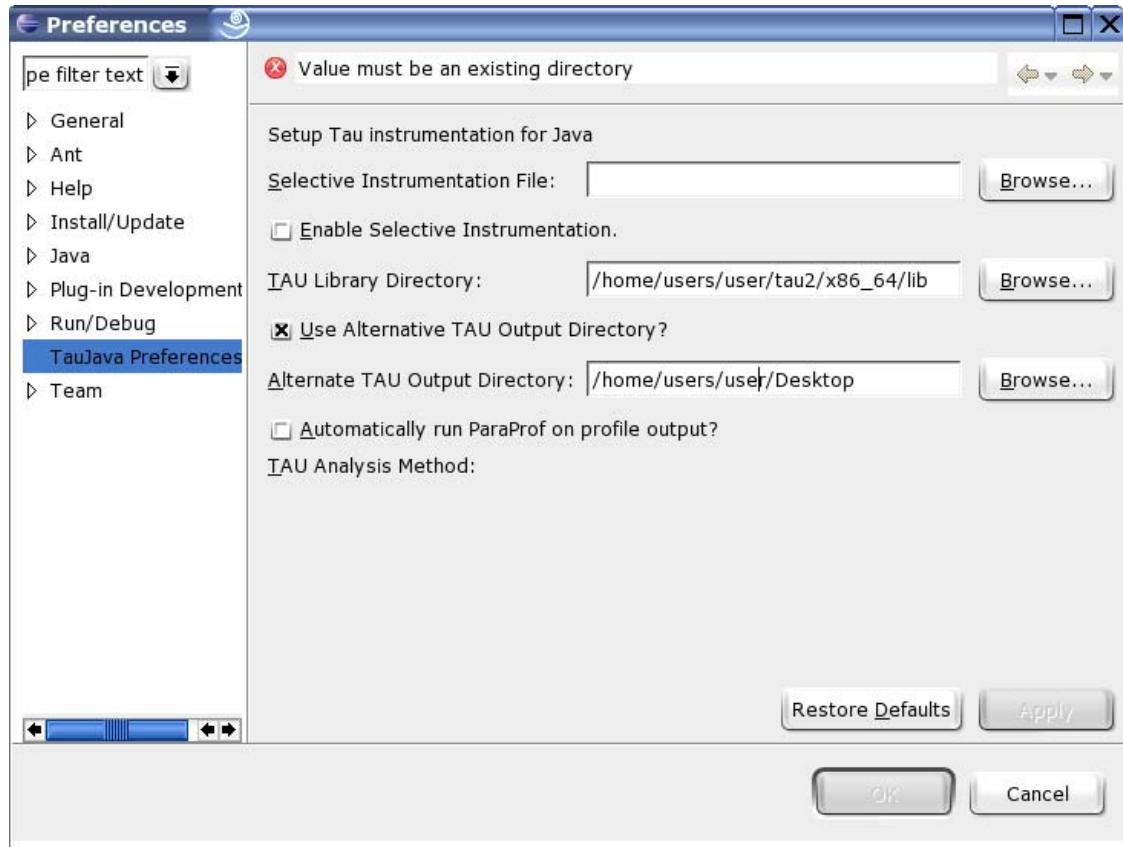
# Chapter 5. Eclipse Tau Java System

## 5.1. Installation

Copy the plugins directory in the tau2/tools/src/taujava directory to the location of your eclipse installation. You may have to restart eclipse if it is running when this is done.

In eclipse go to the Window menu, select Preferences and go to the TauJava Preferences section. Enter the location of the lib directory in the tau installation for your architecture in the Tau Library Directory field. Other options may also be selected at this time.

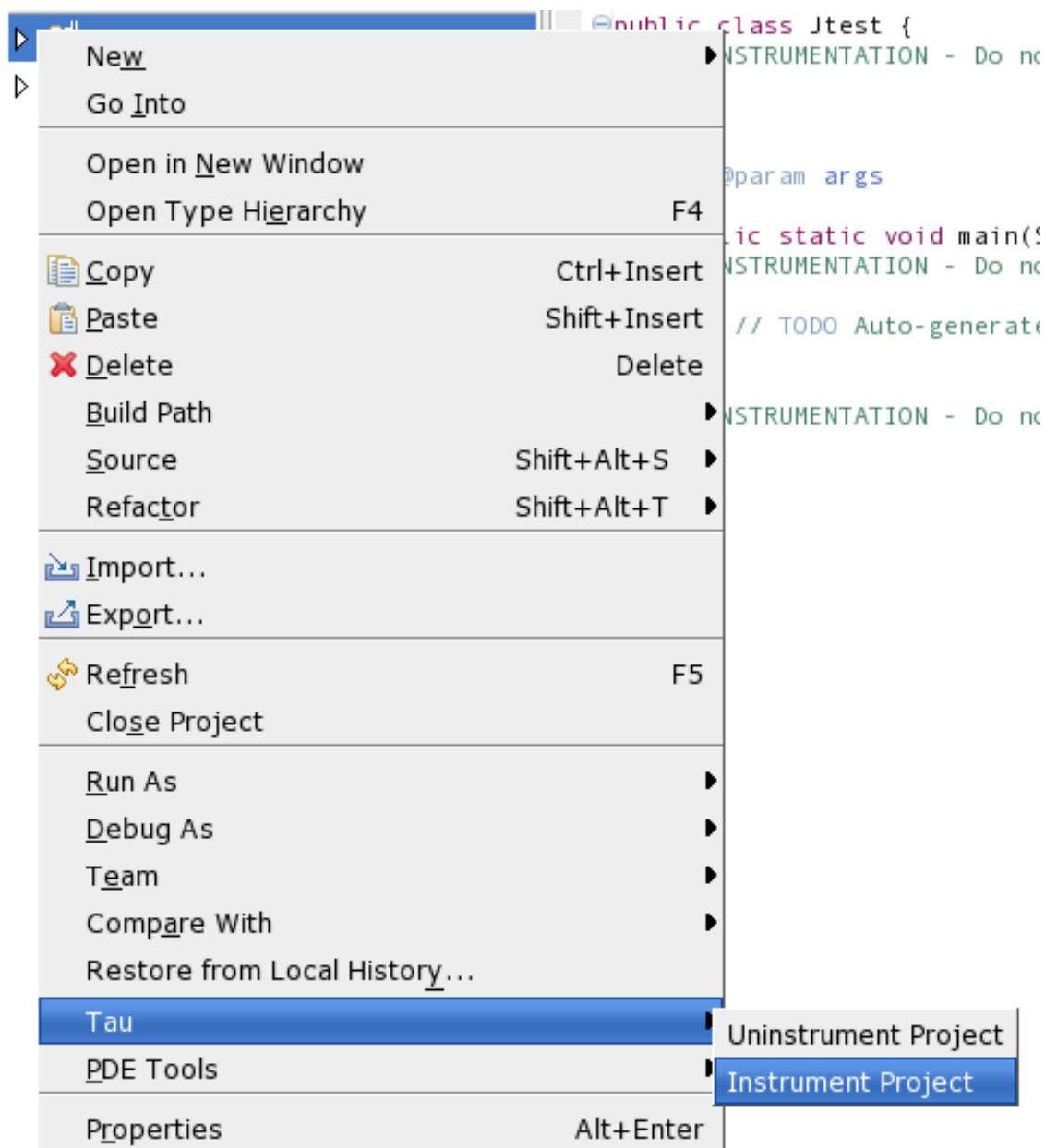
**Figure 5.1. TAUJava Options Screen**



## 5.2. Instrumentation

Java programs can be instrumented at the level of full Java projects, packages or individual Java files. From within the Java view simply right click on the element in the package explorer that you wish to instrument select the Tau pop up menu and click on Instrument Project, Package or Java respectively.

**Figure 5.2. TAUJava Project Instrumentation**



Note that the instrumenter will add the TAU.jar file to the project's class-path the first time any element is instrumented.

Do not perform multiple instrumentations of the same Java file. Do not edit the comments added by the instrumenter or adjust the white space around them. Doing so may prevent the uninstrumenter from working properly.

## 5.3. Uninstrumentation

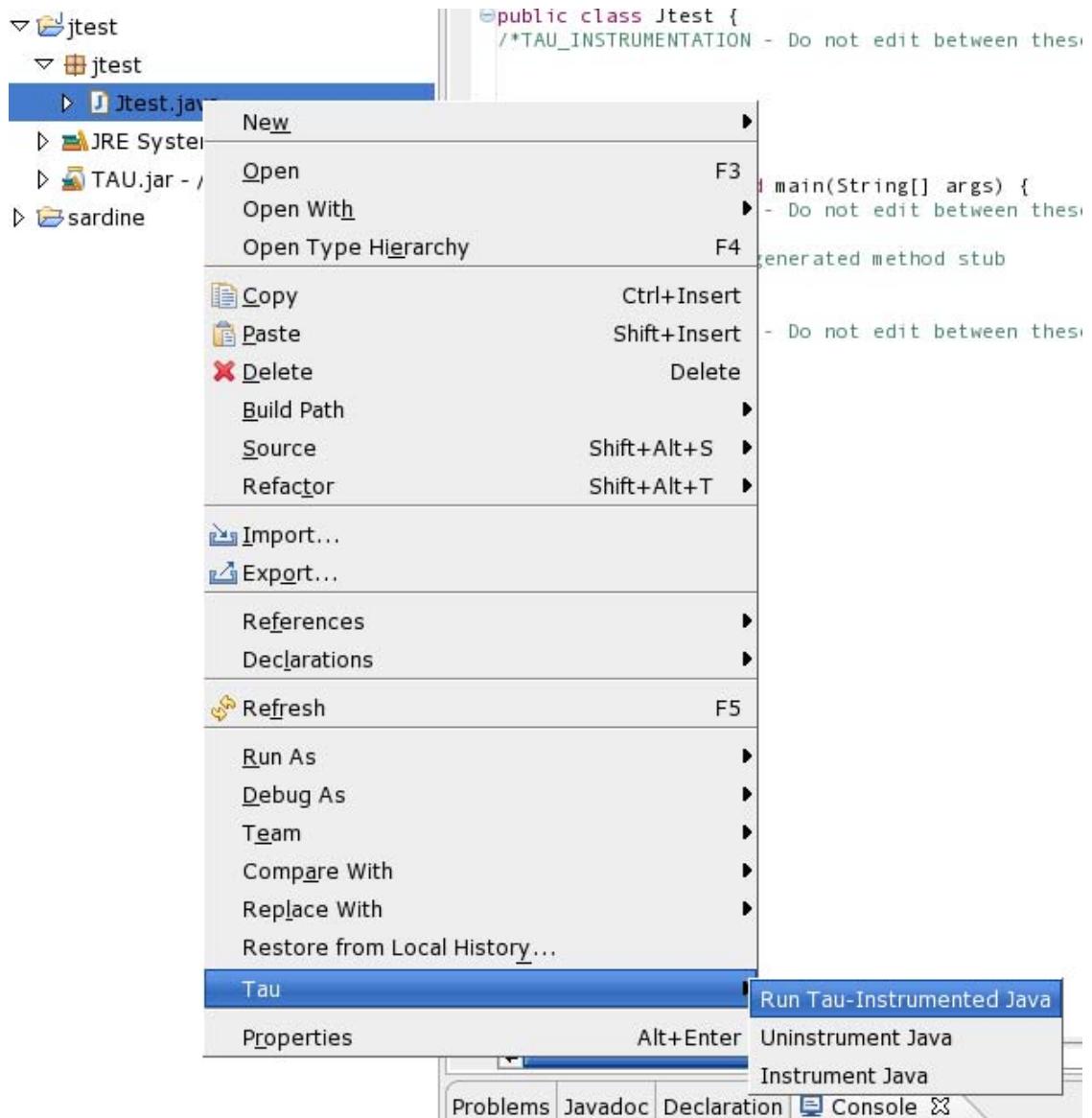
Uninstrumenting a Java project, package or file works just like instrumenting. Just select the uninstrument option instead. Note that the uninstrumenter only removes TAU instrumentation as formatted and commented by the instrumenter. Running the uninstrumenter on code with no TAU instrumentation present has no effect.

## 5.4. Running Java with TAU

To automatically analyze your instrumented project on a Unix-based system TAU must first be configured with the -JDK option, and any other options you want applied to your trace output. On windows the type of analysis to be conducted, Profile, Call path or Trace, should be selected from the Window, Preferences TauJava Preferences menu.

Once that has been accomplished, right click on the Java file containing the main method you want to run, go to the TAU menu and click on Run Tau-Instrumented Java. The program will run and, by default, the profile and/or trace files will be placed in a timestamped directory, inside a directory indicating the name of the file that was run, in the TAU\_Output directory in the home directory of the Java project.

**Figure 5.3. TAUJava Running**



## 5.5. Options

The following options are accessible from the Window, Preferences TAUJava Preferences menu.

Use Alternative TAU Output Directory: Causes the TAU\_Output directory to be placed in the location specified in the associated field. The internal directory structure of the TAU\_Output directory remains unchanged.

Automatically run ParaProf on profile output?: Causes the TAU profile viewer, paraprof, to run on the output of profile and call-path analysis output as soon as the trace files have been produced.

Enable selective instrumentation: Causes Java elements specified in the given selection file to be included or excluded from instrumentation. By default all packages files and methods are included. The file should conform to the TAU file selection format described here.

```
# Any line beginning with a # is a comment and will be disregarded.  
#  
# If an entry is both included and excluded inclusion will take precedence.  
#  
# Entries in INCLUDE or EXCLUDE lists may use * as a wildcard character.  
#  
# If an EXCLUDE_LIST is specified, the methods in the list will not be  
# instrumented.  
#  
BEGIN_EXCLUDE_LIST  
*main*  
END_EXCLUDE_LIST  
#  
# If an INCLUDE_LIST is specified, only the methods in the list will be  
# instrumented.  
#  
BEGIN_INCLUDE_LIST  
*get*  
*set*  
END_INCLUDE_LIST  
#  
# TAU also accepts FILE_INCLUDE/EXCLUDE lists. These may be specified with  
# the wildcard character # to exclude/include multiple files.  
# These options may be used in conjunction with the routine INCLUDE/EXCLUDE  
# lists as shown above.  
#  
BEGIN_FILE_INCLUDE_LIST  
foo.java  
hello#.java  
END_FILE_INCLUDE_LIST  
#  
BEGIN_FILE_EXCLUDE_LIST  
bar.java  
END_FILE_EXCLUDE_LIST  
# Note that the order of the individual sections does not matter  
# and not all of the sections need to be included. Each section  
# must be closed.
```

# Chapter 6. Eclipse PTP / CDT plug-in System

## 6.1. Installation

Be certain that the PTP [http://www.eclipse.org/ptp/downloads.php]/CDT [http://www.eclipse.org/cdt/downloads.php]/Photran [http://www.eclipse.org/photran/download.php] plug-ins are installed and running properly in your eclipse installation. Use Tau's perfdmf\_configure utility to set up a performance database for Eclipse to store profile output.

Run the install\_plug-ins.sh script located in [tau installation]/tools/src/eclipse with the location of your eclipse installation. e.g:  
~/tau2/tools/src/eclipse/install\_plug-ins.sh /opt/eclipse

Restart eclipse with the -clean flag after installing the plugins.

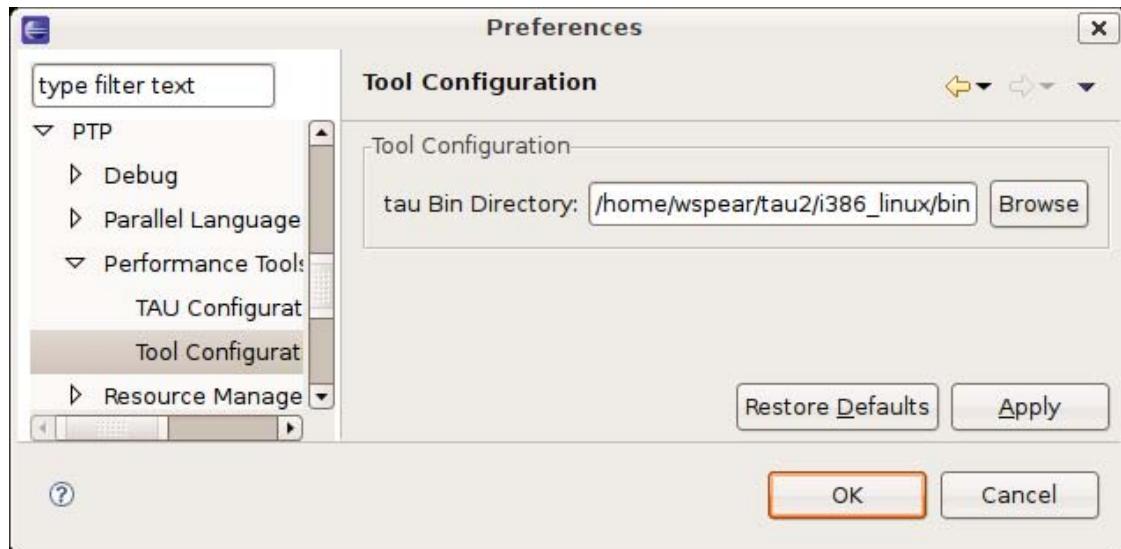


### Note

By default Eclipse will detect the presence of TAU on your system and configure itself appropriately so long as the TAU bin directory is in your path. Only if this fails will you need to setup the TAU preferences manually.

In eclipse go to the Window menu, select Preferences and go to the Performance Tools preferences section and the Tool Configuration subsection. If the PTP is available the Performance Tools section will be under the PTP menu. Enter the location of the desired TAU bin directory in your in the tau Bin Directory field.

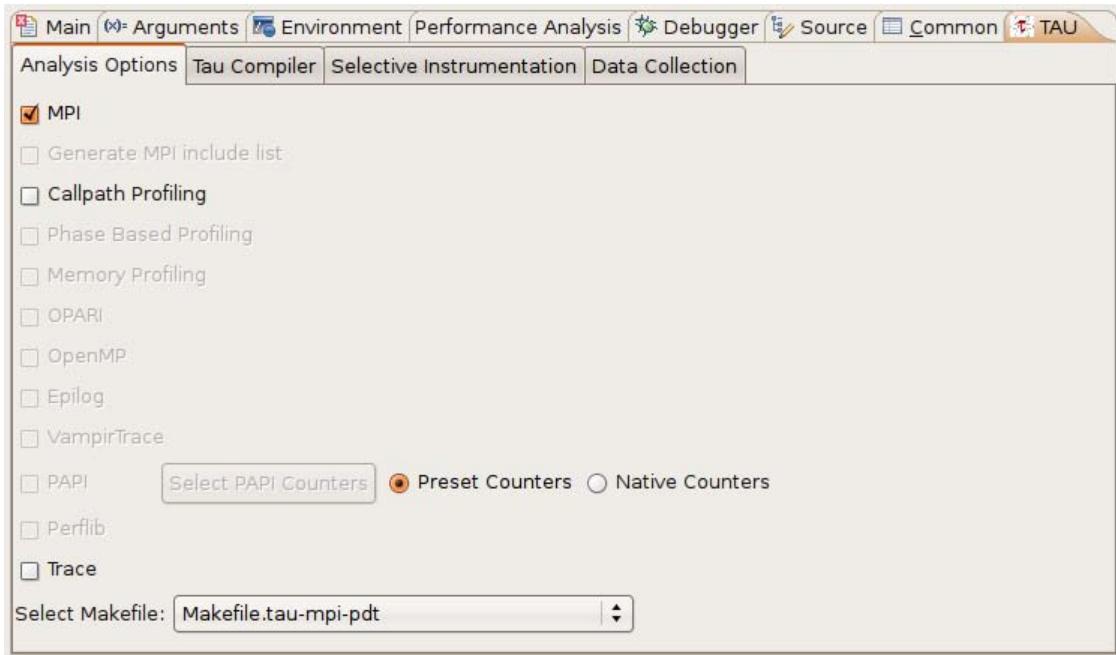
**Figure 6.1. TAU Setup**



## 6.2. Creating a Tau Launch Configuration

To create a TAU launch configuration, click the profile button added near the run and debug buttons. This will provide an interface for launching either a standard or parallel C, C++ or Fortran application, similar to the interface provided by the standard run configuration dialog. You may select a pre-existing run configuration or create a new one in the usual way.

**Figure 6.2. TAU Launch Configuration**



The run configuration options are equivalent to those of a standard run configuration, with the addition of a performance analysis tab a parametric study tab and a TAU tab. To run an application with TAU first make sure that the TAU option is selected in the drop down box on the performance analysis tab. You may also specify that a Tau instrumented executable should not be run after it is built. This option will leave a new TAU specific build configuration available for your use. It will have the name of the original build configuration, with the tau configuration options used appended. The executables available in such build configurations can be run through the standard run and debug launch configurations. This option can be useful if you need to launch Tau instrumented binaries outside of eclipse. There is also an option to select existing performance data. This will upload data specified on the filesystem to a selected database, rather than generating the data from a project in Eclipse.

On the TAU tab you must select a Tau makefile from the available makefiles in the Tau architecture directory you specified. You may select specific configuration options to narrow the list of makefiles in the dropdown box. Only makefiles configured with the -pdt option will be listed. Additional Tau compiler options are provided on the Tau Compiler sub-tab.

If you select a makefile with the PAPI counter library and -MULTIPLECOUNTERS enabled you may specify the PAPI environment variables using the Select PAPI Counters button. The counters you select will be placed in the environment variables list for your run configuration.

You may specify the use of TAU selective instrumentation either by selecting a pre-defined selective instrumentation file, by selecting the internal option to have Tau to use a file generated by the selective instrumentation commands available in the Eclipse workspace or by selecting the automatic option to have eclipse generate a selective instrumentation file using TAU's tau\_reduce utility. Note that the automatic option will cause your project to be rebuilt and run twice.

By default TAU profile data will only be stored in a perfdmf database, if available. The database may be selected on the Data Collection sub-tab. You may specify that performance data should be kept on the file-system with the Keep Profiles option.

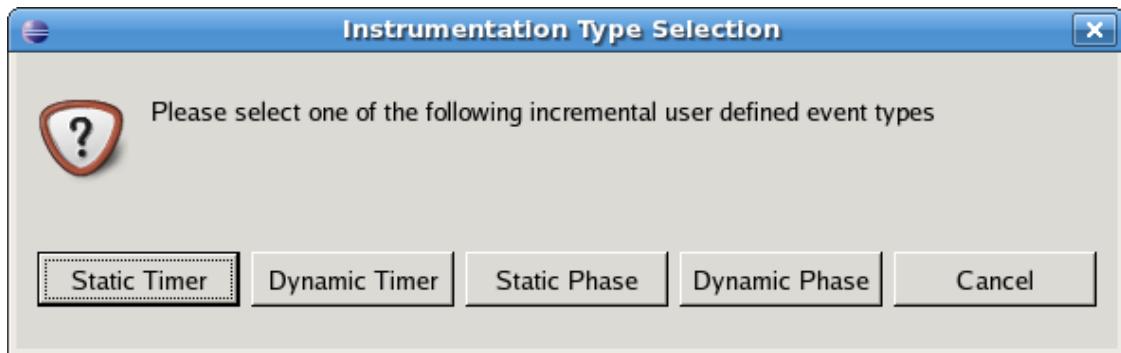
If you wish to collect the resulting profile data on TAU's online Portal [<http://tau.nic.uoregon.edu>], check the "Upload profile data to TAU Portal" box. After the profiling has finished you will be prompted to provide your user name, password and specify the destination workspace. To view the profile data log on to the portal and select the specified workspace.

## 6.3. Selective Instrumentation

C, C++ and Fortran programs have several selective instrumentation options in Eclipse. The selective instrumentation sub-menu of the right click menu provided by C/C++ and Fortran projects, source files and routines in the C/C++ and program outline views allows inclusion, exclusion and loop level instrumentation to be specified for each of these objects. You may also clear instrumentation specified for each of these levels from the selective instrumentation menu.

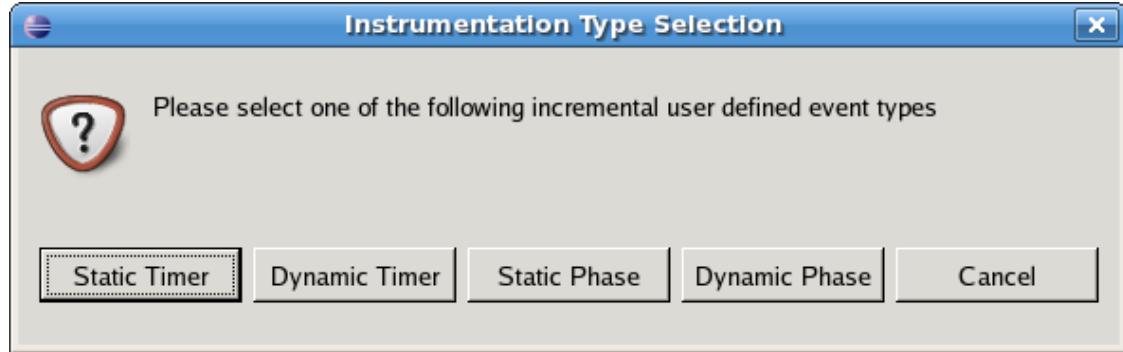
The source editor's context menu allows the insertion of interval and atomic user defined events. To specify an atomic user defined event, place the cursor on the line where you want the event to trigger, right click, go to the Selective Instrumentation sub-menu and select Insert TAU Atomic User Defined Event. Put the name you wish to associate with the event in the first context window that appears. Put either a numeric constant or the name of a valid numeric variable in the second window.

**Figure 6.3. Optional User Defined Events**



To specify an interval based user defined event, select the source code you wish to be included in the interval, right click, go to the Selective Instrumentation sub-menu and select Insert TAU Interval (start/stop) User Defined Event. You may select use of a Static Timer, Dynamic Timer, Static Phase or Dynamic Phase event. Note that to get phase data you must select a Tau makefile configured with the -PROFILEPHASE option. Once you have selected the event type you will be prompted to enter a name for the event.

**Figure 6.4. Adding User Defined Events**



All selective instrumentation options are placed in the tau.selective file in your project's main directory. This file is automatically employed when the Tau launch configuration has "internal" selective instrumentation selected. You may safely edit this file manually so long as it remains a valid Tau selective instrumentation file.

## 6.4. Launching a Program and Collecting Data

To launch your project with Tau either select the Profile button from the profile launch configuration window, select your launch configuration from the dropdown menu of the profile button or, if your desired configuration is already selected, simply click on the profile button.

If a perfdmf database is configured and available, Tau profile data will be saved there. Trace data and other performance data output will be stored in your project's top level directory. If a perfdmf database is not available or you have selected to save profile data on the file system profile output will appear in a Profiles directory in your project's top level directory. Profiles are organized in sub-directories by the Tau configuration options used to generate them and the time-stamp of their creation.

---

# **Chapter 7. Tools**

## Name

tau\_compiler.sh -- Instrumenting source files.

```
tau_compiler.sh [ -p profile ] [ -optVerbose ] [ -optQuiet ] [ -optPdtDir=dir ] [ -optPdtF95Opts=opts ] [ -optPdtF95Reset=opts ] [ -optPdtCOptions=opts ] [ -optPdtCReset=opts ] [ -optPdtF90Parser=parser ] [ -optGnuFortranParser ] [ -optGnuCleanscapeParser ] [ -optPdtUser=opts ] [ -optTauInstr=path ] [ -optDetectMemoryLeaks ] [ -optIncludeMemory ] [ -optPreProcess ] [ -optCPP=path ] [ -optCPPOpts=options ] [ -optCPPReset=options ] [ -optTauSelectFile=file ] [ -optPDBFile=file ] [ -optTau=opts ] [ -optCompile=opts ] [ -optTauDefs=opts ] [ -optTauIncludes=opts ] [ -optReset=opts ] [ -optLinking=opts ] [ -optLinkReset=opts ] [ -optTauCC=cc ] [ -optOpariTool=path/opari ] [ -optOpariDir=path ] [ -optOpariOpts=opts ] [ -optOpariReset=opts ] [ -optOpari2Tool=path/opari2 ] [ -optOpari2ConfigTool=path/opari2_config ] [ -optOpari2Dir=path ] [ -optOpari2Opts=opts ] [ -optOpari2Reset=opts ] [ -optNoMpi ] [ -optMPI ] [ -optNoRevert ] [ -optRevert ] [ -optKeepFiles ] [ -optAppC ] [ -optAppCXX ] [ -optAppF90 ] [ -optShared ] [ -optCompInst ] [ -optPDTInst ] [ -optDisableHeaderInst ] { compiler } [ compiler_options ] [ -optTauWrapFile=filename ]
```

## Description

The TAU Compiler provides a simple way to automatically instrument an entire project. The TAU Compiler can be used on C, C++, fixed form Fortran, and free form Fortran.

## Options

-optVerbose Turn on verbose debugging messages.

-optQuiet Suppresses excessive output.

-optDetectMemoryLeaks Instructs TAU to detect any memory leaks in C/C++ programs.TAU then tracks the source location of the memory leak as wellas the place in the callstack where the memory allocation wasmade.

-optPdtDir=<*dir*> The PDT architecture directory. Typically \$(PDTDIR) / \$(PDTARCHDIR).

-optPdtF95Opts=<*opts*> Options for Fortran parser in PDT (f95parse).

-optPdtF95Reset=<*opts*> Reset options to the Fortran parser to the given list.

-optPdtCOpts=<*opts*> Options for C parser in PDT (cparse). Typically \$(TAU\_MPI\_INCLUDE) \$(TAU\_INCLUDE) \$(TAU\_DEFS).

-optPdtCReset=<*opts*> Reset options to the C parser to the given list

-optPdtCxxOpts=<*opts*> Options for C++ parser in PDT (cxxparse). Typically \$(TAU\_MPI\_INCLUDE) \$(TAU\_INCLUDE) \$(TAU\_DEFS).

-optPdtCxxReset=<*opts*> Reset options to the C++ parser to the given list

-optPdtF90Parser=<*parser*> Specify a different Fortran parser. For e.g., f90parse instead of f95parse.

-optGnuFortranParser=<*parser*> Specify the GNU gfortran Fortran parser gfparsen instead of f95parse

-optGnuCleanscapeParser Uses the Cleanscape Fortran parser f95parse instead of GNU's gfortran.

-optPdtUser=<opts> Optional arguments for parsing source code.

-optTauInstr=<path> Specify location of tau\_instrumentor. Typically \$(TAUROOT)/\$(CONFIG\_ARCH)/bin/tau\_instrumentor.

-optIncludeMemory For internal use only.

-optPreProcess Preprocess the source code before parsing. Uses /usr/bin/cpp-P by default.

-optCPP=<path> Specify an alternative preprocessor and pre-process the sources.

-optCPPOpts=<options> Specify additional options to the C pre-processor.

-optCPPReset=<options> Reset C preprocessor options to the specified list.

-optTauSelectFile=<file> Specify selective instrumentation file for tau\_instrumentor.

-optPDBFile=<file> Specify PDB file for tau\_instrumentor. Skips parsing stage.

-optTau=<opts> Specify options for tau\_instrumentor.

-optCompile=<opts> Options passed to the compiler. Typically \$(TAU\_MPI\_INCLUDE) \$(TAU\_INCLUDE) \$(TAU\_DEFS) .

-optTauDefs=<opts> Options passed to the compiler by TAU. Typically \$(TAU\_DEFS) .

-optTauIncludes=<opts> Options passed to the compiler by TAU. Typically \$(TAU\_MPI\_INCLUDE) \$(TAU\_INCLUDE) .

-optReset=<opts> Reset options to the compiler to the given list.

-optLinking=<opts> Options passed to the linker. Typically \$(TAU\_MPI\_FLIBS) \$(TAU\_LIBS) \$(TAU\_CXXLIBS) .

-optLinkReset=<opts> Reset options to the linker to the given list.

-optTauCC=<cc> Specifies the C compiler used by TAU.

-optOpariTool=<path/o pari> Specifies the location of the Opari tool.

-optOpariDir=<path> Specifies the location of the Opari directory.

-optOpariOpts=<opts> Specifies optional arguments to the Opari tool.

-optOpariReset=<opts> Resets options passed to the Opari tool.

-optNoMpi Removes -l\*mpi\* libraries during linking (default).

-optMpi Does not remove -l\*mpi\* libraries during linking.

-optNoRevert Exit on error. Does not revert to the original compilation rule on error.

-optRevert Revert to the original compilation rule on error (default).

-optKeepFiles Does not remove intermediate .pdb and .inst.\* files.

-optAppCC Sets the failsafe C compiler.

-optAppCXX Sets the failsafe C++ compiler.

-optAppF90 Sets the failsafe F90 compiler

-optShared Use shared library version of TAU

-optCompInst Use compiler-based instrumentation

-optNoCompInst Do not revert to compiler instrumentation if source instrumentation fails.

-optPDTInst Use PDT-based instrumentation

-optHeaderInst Enable instrumentation of headers

-optDisableHeaderInst Disable instrumentation of headers

-optTrackIO Specify wrapping of POSIX I/O calls at link time.

-optWrappersDir=" " Specify the location of the link wrappers directory.

-optTauUseCXXForC Specifies the use of a C++ compiler for compiling C code

-optTauWrapFile=<filename> Specify path to the link\_options.tau file generated by tau\_wrap

-optFixHashIf

## Name

```
vtf2profile -- Generate a TAU profile set from a vampir trace file  
vtf2profile [ -p profile ] [ -i interval_start interval_end ] [ -c ] [ -h ] { -f  
tracefile }
```

## Description

vtf2profile is created when TAU is configured with the -vtf=<vtf\_dir> option. This tool converts a VTF trace file (\*.vpt) to a tau profile set (profile.A.B.C where A, B and C are the node, context and thread numbers respectively).

The vtf file to be read is specified in the command line by the -f flag followed by the file's location. The VTF tracefile specified may be in gzipped form, eg app.vpt.gz. -p is similarly used to specify the relative path to the directory where the profile files should be stored. If no output directory is specified the current directory will be used. A contiguous interval within the vtf file may be selected for conversion by using the -i flag followed by two integers, representing the timestamp of the start and end of the desired interval respectively. The entire vtf file is converted if no interval is given.

## Options

- f *tracefile* -Specify the Vampir tracefile to be converted.
- p *profile* -Specify the location where the profile file(s) should be written.
- i *interval\_start interval\_end* -Limit the profile produced to the specified interval within the vampir trace file.
- c -Opens a command line interface for the program.
- h -Displays a help message.

## Examples

To convert a vampir tracefile, trace.vpt, to an equivalent TAU profile, use the following:

```
vtf2profile -f trace.vpt
```

To produce a TAU profile in the ./profiles directory representing only the events from the start of the tracefile to timestamp 6000, use:

```
vtf2profile -f trace.vpt -p ./profiles -i 0 6000
```

## See Also

[tau2vtf](#), [tau2profile](#)

## Name

```
tau2vtf -- convert TAU tracefiles to vampir tracefiles  
  
tau2vtf [ -nomessage ] [ -v ] [[ -a ] | [ -fa ]] { tau_tracefile } { tau_eventfile } {  
vtf_tracefile }
```

## Description

This program is generated when TAU is configured with the -vtf=<vtf\_dir> option.

The tau2vtf trace converter takes a single tau\_tracefile (\*.trc) and tau\_eventfile (\*.edf) and produces a corresponding vtf\_tracefile (\*.vtf). The input files and output file must be specified in that order. Multi-file TAU traces must be merged before conversion.

The default output file format is VTF3 binary. If the output filename is given as the .vpt.gz type, rather than .vpt, the output file will be gzipped. There are two additional output format options. The command line argument '-a' produces the vtf file output in ASCII VTF3 format. The command line argument '-fa' produces the vtf file output in the FAST ASCII VTF3 format. Note that these arguments are mutually exclusive.

## Options

- nomessage Suppresses printing of message information in the trace.
- v Verbose mode sends trace event descriptions to the standard output as they are converted.
- a Print the vtf file output in the human-readable VTF3 ASCII format
- fa Print the vtf file in the simplified human-readable FAST ASCII VTF3 format

## Examples

The program must be run with the tau trace, tau event and vtf output files specified in the command line in that order. Any additional arguments follow. The following will produce a VTF, app.vpt, from the TAU trace and event files merged.trc and tau.edf trace file:

```
tau2vtf merged.trc tau.edf app.vpt
```

The following will convert merged.trc and tau.edf to a gzipped FAST ASCII vampir tracefile app.vpt.gz, with message events omitted:

```
tau2vtf merged.trc tau.edf app.vpt.gz -nomessage -fa
```

## See Also

vtf2profile, tau2profile, tau\_merge, tau\_convert

## Name

```
tau2profile -- convert TAU tracefiles to TAU profile files  
tau2vprofile [ -d directory ] [ -s snapshot_interval ] { tau_tracefile } {  
  tau_eventfile }
```

## Description

This program is generated when TAU is configured with the -TRACE option.

The tau2profile converter takes a single tau\_tracefile (\*.trc) and tau\_eventfile (\*.edf) and produces a corresponding series of profile files. The input files must be specified in that order, with optional parameters coming afterward. Multi-file TAU traces must be merged before conversion.

## Options

- d Output profile files to the specified 'directory' rather than the current directory.
- s Output a profile snapshot showing the state of the profile data accumulated from the trace every 'snapshot\_interval' time units. The snapshot profiles are placed sequentially in directories labeled 'snapshot\_n' where 'n' is an integer ranging from 0 to the total number of snapshots -1.

## Examples

The program must be run with the tau trace and tau event files specified in the command line in that order. Any additional arguments follow. The following will produce a profile file array, from the TAU trace and event files merged.trc and tau.edf trace file:

```
tau2profile merged.trc tau.edf
```

The following will convert merged.trc and tau.edf to a series of profiles one directory higher. It will also produce a profile snapshot every 250,000 time units:

```
tau2profile merged.trc tau.edf -d ./.. -s 250000
```

## See Also

vtf2profile, tau2vtf, tau2otf, tau\_merge, tau\_convert

## Name

tau2elg -- convert TAU tracefiles to Epilog tracefiles

```
tau2elg [ -nomessage ] [ -v ] { tau_tracefile } { tau_eventfile } { elg_tracefile }
```

## Description

This program is generated when TAU is configured with the -epilog=<epilog\_dir> option.

The tau2elg trace converter takes a tau trace file (\*.trc) and event definition file (\*.edf) and produces a corresponding epilog binary trace file (\*.elg). Multi-file TAU traces must be merged before conversion.

## Options

-nomessage Suppresses printing of message information in the trace.

-v Verbose mode sends trace event descriptions to the standard output as they are converted.

## Examples

The program must be run with the tau trace, tau event and elg output files specified in the command line in that order. Any additional arguments follow. The following would convert merged.trc and tau.edf to the Epilog tracefile app.elg, with message events omitted:

```
./tau2vtf merged.trc tau.edf app.elg -nomessage
```

## See Also

[tau\\_merge](#)

## Name

tau2slog2 -- convert TAU tracefiles to SLOG2 tracefiles

```
tau2slog2 [ options ] { tau_tracefile } { tau_eventfile } { -o output.slog2 }
```

## Description

This program is generated when TAU is configured with the -slog2 or -slog2=<slog2\_dir> option.

The tau2slog2 trace converter takes a single tau trace file (\*.trc) and event definition file (\*.edf) and produces a corresponding slog2 binary trace file (\*.slog2).

The tau2slog2 converter is called from the command line with the locations of the tau trace and event files. These arguments must be followed by the -o flag and the name of the slog2 file to be written. tau2slog 2 accepts no other arguments.

## Options

[ -h | --h | -help | --help ] Display HELP message.

[ -tc ] Check increasing endtime order, exit when 1st violation occurs.

[ -tcc ] Check increasing endtime order, continue when violations occur.

[ -nc number ] Number of children per node (default is 2)

[ -ls number ] Max byte size of leaf nodes (default is 65536)

[ -o output.slog2 ] Output filename with slog2 suffix

## Examples

A typical invocation of the converter, to create app.slog2, is as follows:

```
tau2slog2 app.trc tau.edf -o app.slog2
```

## See Also

tau\_merge, tau\_convert

## Name

tau2otf -- convert TAU tracefiles to OTF tracefiles for Vampir/VNG

tau2otf [ -n streams ] [ -nomessage ] [ -v ]

## Description

This program is generated when TAU is configured with the -otf=<otf\_dir> option. The tau2otf trace converter takes a TAU formatted tracefile (\*.trc) and a TAU event description file (\*.edf) and produces an output trace file in the Open Trace Format (OTF). The user may specify the number of output streams for OTF. The input files and output file must be specified in that order. TAU traces should be merged using tau\_merge prior to conversion.

## Options

-n streams Specifies the number of output streams (default is 1). -nomessage Suppresses printing of message information in the trace. -v Verbose mode sends trace event descriptions to the standard output as they are converted.

## Examples

The program must be run with the tau trace, tau event and otf output files specified in the command line in that order. Any additional arguments follow. The following will produce an OTF file, a pp.otf and other related event and definition files, from the TAU trace and event files merged.trc and tau.edf trace file:

```
tau2otf merged.trc tau.edf app.otf
```

## See Also

tau2vtf(1), tau2profile(1), vtf2profile(1), tau\_merge(1), tau\_convert(1)

## Name

perf2tau -- converts PerfLib profiles to TAU profile files

`perf2tau { data_directory } [ -h ] [ -flat ]`

## Description

Converts perflib data to TAU format.

If an argument is not specified, it checks the `perf_data_directory` environment variable. Then opens `perf_data.timing` directory to read perflib data. If no args are specified, it tries to read `perf_data.<current_date>` file.

## Options

`-h` Display the help information.

`-flat` Suppresses callpath profiles, each callpath profile will be flattened to show only the function profile.

## Examples

```
%> perf2tau timing
```

## See Also

`vtf2profile`, `tau2vtf`, `tau2otf`, `tau_merge`, `tau_convert`

## Name

tau\_merge -- combine multiple node and or thread TAU tracefiles into a merged tracefile

```
tau_merge [ -a ] [ -r ] [ -n ] [ -e eventfile_list ] [ -m output_eventfile ] { tracefile_list } [ { output_tracefile } | { - } ]
```

## Description

tau\_merge is generated when TAU is configured with the -TRACE option.

This tool assembles a set of tau trace and event files from multiple multiple nodes or threads across a program's execution into a single unified trace file. Many TAU trace file tools operate on merged trace files.

Minimally, tau\_merge must be invoked with a list of unmerged trace files followed by the desired name of the merged trace file or the - flag to send the output to the standard out. Typically the list can be designated by giving the shared name of the trace files to be merged followed by desired range of thread or node designators in brackets or the wild card character '\*' to encompass variable thread and node designations in the filename (trace.A.B.C.trc where A, B and C are the node, context and thread numbers respectively). For example taurace.\*.trc would represent all tracefiles in a given directory while taurace.[0-5].0.0.trc would represent the tracefiles of nodes 0 through 5 with context 0 and thread 0.

tau\_merge will generate the specified merged trace file and an event definition file, tau.edf by default.

The event definition file can be given an alternative name by using the '-m' flag followed by the desired filename. A list of event definition files to be merged can be designated explicitly by using the '-e' flag followed by a list of unmerged .edf files, specified in the same manner as the trace file list.

If computational resources are insufficient to merge all trace and event files simultaneously the process may be undertaken hierarchically. Corresponding subsets of the tracefiles and eventfiles may be merged in sequence to produce a smaller set of files that can then be merged into a singular fully merged tracefile and eventfile. E.g. for a 100 node trace, trace sets 1-10, 11-20, ..., 91-100 could be merged into traces 1a, 2a, ..., 10a. Then 1a-10a could be merged to create a fully merged tracefile.

## Options

- e eventfile\_list explicitly define the eventfiles to be merged
- m output\_eventfile explicitly name the merged eventfile to be created
- send the merged tracefile to the standard out
- a adjust earliest timestamp time to zero
- r do not reassemble long events
- n do not block waiting for new events. By default tau\_merge will block and wait for new events to be appended if a tracefile is incomplete. This command allows offline merging of (potentially) incomplete tracefiles.

## Examples

To merge all TAU tracefiles into app.trc and produce a merged tau.edf eventfile:

```
tau_merge *.trc app.trc
```

To merge all eventfiles 0-255 into ev0\_255merged.edf and TAU tracefiles for nodes 0-255 into the standard out:

```
tau_merge -e events.[0-255].edf -m ev0_255merged.edf \
tautrace.[0-255].*.trc -
```

To merge eventfiles 0, 5 and seven info ev057.edf and tau tracefiles for nodes 0, 5 and 7 with context and thread 0 into app.trc:

```
tau_merge -e events.0.edf events.5.edf events.7.edf -m ev057.edf \
tautrace.0.0.0.trc tautrace.5.0.0.trc tautrace.7.0.0.trc app.trc
```

## See Also

[tau\\_convert](#)

[tau2profile](#)

[tau2vtf](#)

[tau2elg](#)

[tau2slog2](#)

## Name

tau\_treemerge.pl -- combine multiple node and or thread TAU tracefiles into a merged tracefile

tau\_treemerge.pl [ -n *break\_amount* ]

## Description

tau\_treemerge.pl is generated when TAU is configured with the -TRACE option.

This tool assembles a set of tau trace and event files from multiple nodes or threads across a program's execution into a single unified trace file. Many TAU trace file tools operate on merged trace files.

tau\_treemerge.pl will generate the specified merged trace file and an event definition file, tau.edf by default.

## Options

-n *break\_amount* set the maximum number of trace files to merge in each invocation of tau\_merge. If we need to merge 2000 trace files and if the maximum number of open files specified by unix is 250, tau\_treemerge.pl will incrementally merge the trace files so as not to exceed the number of open file descriptors.

## See Also

[tau\\_merge](#)

[tau\\_convert](#)

[tau2profile](#)

[tau2vtf](#)

[tau2elg](#)

[tau2slog2](#)

## Name

```
tau_convert -- convert TAU tracefiles into various alternative trace formats  
  
tau_convert [[ -alog ] | [ -SSDF ] | [ -dump ] | [ -paraver [-t] ] | [ -pv ] | [ -vampir [ -  
longsymbolbugfix ] [ -compact ] [ -user ] | [ -class ] | [ -all ] ] [ -nocomm ] ] [ out-  
puttrc ] { inputtrc } { edffile }
```

## Description

tau\_convert is generated when TAU is configured with the -TRACE option.

This program requires specification of a TAU tracefile and eventfile. It will convert the given TAU traces to the ASCII-based trace format specified in the first argument. The conversion type specification may be followed by additional options specific to the conversion type. It defaults to the single threaded vampir format if no other format is specified. tau\_convert also accepts specification of an output file as the last argument. If none is given it prints the converted data to the standard out.

## Options

- alog convert TAU tracefile into the alog format (This format is deprecated. The SLOG2 format is recommended.)
- SSDF convert TAU tracefile into the SDDF format
- dump convert TAU tracefile into multi-column human readable text
- paraver convert TAU tracefile into paraver format
- t indicate conversion of multi threaded TAU trace into paraver format
- pv convert single threaded TAU tracefile into vampir format (all -vampir options apply) (default)
- vampir convert multi threaded TAU tracefile into vampir format
- longsymbolbugfix make the first characters of long, similar identifier strings unique to avoid a bug in vampir
- compact abbreviate individual event entries
- all compact all entries (default)
- user compact user entries only
- class compact class entries only
- nocomm disregard communication events
- [outputtrc] specify the name of the output tracefile to be produced

## Examples

To print the contents of a TAU tracefile to the screen:

```
tau_convert -dump app.trc tau.edf
```

To convert a merged, threaded TAU tracefile to paraver format:

```
tau_convert -paraver -t app.trc tau.edf app.pv
```

## See Also

[tau\\_merge](#), [tau2vtf](#), [tau2profile](#), [tau2slog2](#)

## Name

```
tau_reduce -- generates selective instrumentation rules based on profile data  
tau_reduce { -f filename } [ -n ] [ -r filename ] [ -o filename ] [ -v ] [ -p ]
```

## Description

tau\_reduce is an application that will apply a set of user-defined rules to a pprof dump file (**pprof -d**) in order to create a select file that will include an exclude list for selective implementation for TAU. The user must specify the name of the pprof dump file that this application will use. This is done with the **-f** filename flag. If no rule file is specified, then a single default rule will be applied to the file. This rule is: numcalls > 1000000 & usecs/call < 2, which will exclude all routines that are called at least 1,000,000 times and average less than two microseconds per call. If a rule file is specified, then this rule is not applied. If no output file is specified, then the results will be printed out to the screen.

## Rules

Users can specify a set of rules for tau\_reduce to apply. The rules should be specified in a separate file, one rule per line, and the file name should be specified with the appropriate option on the command line. The grammar for a rule is: [GROUPNAME:]FIELD OPERATOR NUMBER. The GROUPNAME followed by the colon (:) is optional. If included, the rule will only be applied to routines that are a member of the group specified. Only one group name can be applied to each rule, and a rule must follow a groupname. If only a groupname is given, then an unrecognized field error will be returned. If the desired effect is to exclude all routines that belong to a certain group, then a trivial rule, such as GROUP:numcalls > -1 may be applied. If a groupname is given, but the data does not contain any groupname data, then an error message will be given, but the rule will still be applied to the data ignoring the groupname specification. A FIELD is any of the routine attributes listed in the following table:

**Table 7.1. Selection Attributes**

ATTRIBUTE NAME	MEANING
numcalls	Number of times the routine is called
numsubrs	Number of subroutines that the routine contains
percent	Percent of total implementation time
usec	Exclusive routine running time, in microseconds
cumusec	Inclusive routine running time, in microseconds
count	Exclusive hardware count
totalcount	Inclusive hardware count
stddev	Standard deviation
usecs/call	Microseconds per call
counts/call	Hardware counts per call

Some FIELDS are only available for certain files. If hardware counters are used, then usec, cumusec, usecs/call are not applicable and an error is reported. The opposite is true if timing data is used rather than hardware counters. Also, stddev is only available for certain files that contain that data.

An OPERATOR is any of the following: < (less than), > (greater than), or = (equals).

A NUMBER is any number.

A compound rule may be formed by using the & (and) symbol in between two simple rules. There is no "OR" because there is an implied or between two separate simple rules, each on a separate line. (ie the compound rule usec < 1000 OR numcalls = 1 is the same as the two simple rules "usec < 1000" and "numcalls = 1").

## Rule Examples

```
#exclude all routines that are members of TAU_USER and have less than
#1000 microseconds
TAU_USER:usec < 1000

#exclude all routines that have less than 1000 microseconds and are
#called only once.
usec < 1000 & numcalls = 1

#exclude all routines that have less than 1000 usecs per call OR have a percent
#less than 5
usecs/call < 1000
percent < 5
```

NOTE: Any line in the rule file that begins with a # is a comment line. For clarity, blank lines may be inserted in between rules and will also be ignored.

## Options

- f filename specify filename of pprof dump file
- p print out all functions with their attributes
- o filename specify filename for select file output (default: print to screen)
- r filename specify filename for rule file
- v verbose mode (for each rule, print out rule and all functions that it excludes)

## Examples

To print to the screen the selective instrumentation list for the paraprof dump file app.prf with default selection rules use:

```
tau_reduce -f app.prf
```

To create a selection file, app.sel, from the paraprof dump file app.prf using rules specified in foo.rlf use:

```
tau_reduce -f app.prf -r foo.rlf -o app.sel
```

## See Also

---

## Name

`tau_ompcheck` -- Completes uncompleted do/for/parallel omp directives

`tau_ompcheck { pdbfile } { sourcefile } [ -o outfile ] [ -v ] [ -d ]`

## Description

Finds uncompleted do/for omp directives and inserts closing directives for each one uncompleted. do/for directives are expected immediately before a do/for loop. Closing directives are then placed immediately following the same do/for loop.

## Options

`pdbfile` A pdbfile generated from the source file you wish to check. This pdbfile must contain comments from which the omp directives are gathered. See `pdbcomment` for information on how to obtain comment from a pdbfile.

`sourcefile` A fortran, C or C++ source file to analyzed.

`-o` write the output to the specified outfile.

`-v`verbose output, will say which directive where added.

`-d` debugging information, we suggest you pipe this unrestrained output to a file.

## Examples

To check file: `source.f90` do: (you will need pdtoolkit/<arch>/bin and tau/utils/ in your path).

```
%>f95parse source.f90  
%>pdbcomment source.pdb > source.comment.pdb  
%>tau_omp source.comment.pdb source.f90 -o source.chk.f90
```

## See Also

`f95parse` `pdbcomment`

## Name

`tau_poe` -- Instruments a MPI application while it is being executed with poe.  
`tau_poe [ -XrunTAUsh- tauOptions ] { application } [ poe options ]`

## Description

This tool dynamically instruments a mpi application by loading a specific mpi library file.

## Options

`tauOptions` To instrument a mpi application a specific TAU library file is loaded when the application is executed. To select which library is loaded use this option. The library files are build according to the options set when TAU is configured. The library file that have been build and thus available for use are in the `[TAU_HOME]/[arch]/lib` directory. The file are listed as `libTAUsh-*.so` where \* is the instrumentation options. For example to use the `libTAUsh-pdt-openmp-opari.so` file let the command line option be `-XrunTAUsh-pdt-openmp-opari`.

## Examples

Instrument a.out with the currently configured options and then run it on four nodes:

```
%>tau_poe ./a.out -procs 4
```

Select the libTAUsh-mpi.so library to instrument a.out with:

```
%>tau_poe -XrunTAUsh-mpi ./a.out -procs 4
```

## Name

tau\_validate -- Validates a TAU installation by performing various tests on each TAU stub Makefile

tau\_validate [ -v ] [ --html ] [ --build ] [ --run ] [ --tag ] { arch directory }

## Description

tau\_validate will attempt to validate a TAU installation by performing various tests on each TAU stub Makefile. Some degree of logic exists to know where a given test applies to a given makefile, but it's not perfect.

## Options

v Verbose output

html Output results in HTML

build Only build

run Only run

tag Only check configurations containing the tag. ie. --tag papi checks only libraries with the -papi in their name.

arch directory Specify an arch directory (e.g. rs6000), or the lib directory (rs6000/lib), or a specific makefile. Relative or absolute paths are ok.

## Example

There is a few examples:

```
bash : ./tau_validate --html x86_64 &> results.html
tcsh : ./tau_validate --html x86_64 >& results.html
```

## Name

tauex -- Allows you to choose a tau configuration at runtime

`tauex [OPTION] [--] { executable } [executable options]`

## Description

Use this script to dynamically load a TAU profiling/tracing library or to select which papi events/domain to use during execution of the application. At runtime tauex will set the LD\_LIBRARY\_PATH and pass any other parameters (or papi events) to the program and execute it with the specified TAU measurement options.

## Options

-d	Enable debugging output, use repeatedly for more output.
-h	Print help message.
-i	Print information about the host machine.
-s	Dump the shell environment variables and exit.
-U	User mode counts
-K	Kernel mode counts
-S	Supervisor mode counts
-I	Interrupt mode counts
-l	List events
-L <event>	Describe event
-a	Count all native events (implies -m)
-n	Multiple runs (enough runs of exe to gather all events)
-e <event>	Specify PAPI preset or native event
-T <option>	Specify TAU option
-v	Debug/Verbose mode
-XrunTAU-<options>	specify TAU library directly

## Notes

Defaults if unspecified: -U -T MPI,PROFILE -e P\_WALL\_CLOCK\_TIME MPI is assumed unless SERIAL is specified PROFILE is assumed unless one of TRACE, VAMPIRTRACE or EPILOG is specified P\_WALL\_CLOCK\_TIME means count real time using fastest available timer

## Example

```
mpirun -np 2 tauex -e PAPI_TOT_CYC -e PAPI_FP_OPS -T MPI,PROFILE --  
./ring
```

## Name

tau\_exec -- TAU execution wrapping script  
tau\_exec [ *options* ] [--] { *exe* } [ *exe options* ]

## Description

Use this script to perform memory or IO tracking on either an instrumented or uninstrumented executable.

## Options

-v	verbose mode
-qsub	BG/P qsub mode
-io	track io
-memory	track memory
-cuda	track GPU events via CUDA (Must be configured with -cuda=<dir>)
-cupti	track GPU events via Nvidia's CUPTI interface (Must be configured with -cupti=<dir>)
-opencl	track GPU events via OpenCL
-armci	track ARMCI events via PARMCI (Must be configured with -armci=<dir>)
-ebs	enable Event-based sampling. See README.sampling for more information
-ebs_period=<count>	sampling period (default 1000)
-ebs_source=<counter>	sets sampling metric (default "itimer")
-T<option>	: specify TAU option
-loadlib=<file.so>	: specify additional load library
-XrunTAU-<options>	specify TAU library directly

## Notes

Defaults if unspecified: -T MPI. MPI is assumed unless SERIAL is specified

CUDA kernel tracking is included, if A CUDA SYNC call is made after each kernel launch and cudaThreadExit( ) is called before the exit of each thread that uses CUDA.

OPENCL kernel tracking is included, if A OPENCL SYNC call is made after each kernel launch and clReleaseContext( ) is called before the exit of each thread that uses CUDA.

## Examples

---

```
mpirun -np 2 tau_exec -io ./ring  
mpirun -np 8 tau_exec -ebs -ebs_period=1000000 -ebs_source=PAPI_FP_INS  
./ring
```

## Name

tau\_timecorrect -- Corrects and reorders the records of tau trace files.

```
tau_timecorrect { trace input file } { EDF input file } { trace output file } { EDF output file }
```

## Description

This program takes in tau trace files, reorders and corrects the times of these records and then outputs the records to new trace files. The time correction algorithm uses a logical clock algorithm with amortization. This is done by adjusting the times of events such that the product of an effect happens after the cause of that effect.

## Options

```
trace input file  
EDF input file  
trace output file  
EDF output file
```

## Name

tau\_throttle.sh -- This tool generates a selective instrumentation file (called throttle.tau) from a program output that has "Disabling" messages.

tau\_throttle.sh

## Description

This tool auto-generates a selective instrumentation file based on output from a program that has the profiling of some of its functions throttled.

## Name

`tau_portal.py` -- This tool is design to interact with the TAU web portal (<http://tau.nic.uoregon.edu>). There are commands for uploading or downloading packed profile files form the TAU portal.

```
tau_portal.py [-help] [--help] {command} {options} [argument]
```

## Description

Each command will initiate a transfer to profile data between the TAU portal and either the filesystem (to be stored as ppk file) or to a PerfDMF database. See `tau_portal --help` for more information.

## Name

`perfdmf_configure` -- Configuration program for a PerfDMF database.

`perfdmf_configure [-h,--help] [--create-default] [-g, --configFile configFile] [-c, --config configuration_name] [-t, --tauroot path]`

## Description

This configuration script will create a new perfdmf database.

## Options

`-h, --help` show help

`--create-default` creates a H2 database with all the default values

`-g, --configFile configFile` specify the path to the file that defines the perfdmf configuration.

`-c, --config configuration_name` specify the name of the perfdmf configuration -c foo is equivalent to -g <home>/ .ParaProf/perfdmf.cfg.foo.

`-t, --tauroot path` Path to the root directory of tau.

## Name

perfdmf\_createapp -- Command line tool to create a application in the perfdmf database.

```
perfdmf_createapp [-h, --help] [-g, --configFile configFile] [-c, --config configuration_name] [-a, --applicationid applicationID] {-n, --name name}
```

## Description

This script will create a new application in the perfdmf database.

## Options

-g, --configFile *configFile* specify the path to the file that defines the perfdmf configuration.

-c, --config *configuration\_name* specify the name of the perfdmf configuration -c foo is equivalent to -g <home>/ .ParaProf/perfdmf.cfg.foo.

-a, --applicationid *applicationID* specify the id number of the newly added application (default uses auto-increment).

-n, --name *name* the name of the application.

## Name

`perfdmf_createexp` -- Command line tool to create a experiment in the perfdmf database.

`perfdmf_createexp [-h, --help] [-g, --configFile configFile] [-c, --config configuration_name] {-a, --applicationid applicationID} {-n, --name name}`

## Description

This script will create a new experiment in the perfdmf database.

## Options

`-g, --configFile configFile` specify the path to the file that defines the perfdmf configuration.

`-c, --config configuration_name` specify the name of the perfdmf configuration -c foo is equivalent to -g <home>/ParaProf/perfdmf.cfg.foo.

`-a, --applicationid applicationID` specify the id number of the application to associate with the new experiment.

`-n, --name name` the name of the application.

## Name

`perfdmf_loadtrial` -- Command line tool to load a trial into the perfdmf database.

```
perfdmf_loadtrial {-a appName} {-x experimentName} {-n name} [options]
```

## Description

This script will create a new trial in the perfdmf database.

## Options

- n, --name *name* the name of the application.
- a, --applicationname *name* specify associated application name for this trial
- x, --experimentname *experimentName* specify the name of the experiment to associate with newly uploaded trial.
- e, --experimentid *experimentID* specify the id number of the experiment to associate with the new trial.
- g, --configFile *configFile* specify the path to the file that defines the perfdmf configuration. (overrides -c)
- c, --config *configuration\_name* specify the name of the perfdmf configuration -c foo is equivalent to -g <.
- t, --trialid *experimentID* specify the id number of the newly uploaded trial.
- m, --metadata *filename* specify the filename of the XML metadata for this trial.
- f, --filetype *filetype* Specify type of performance data, options are: profiles (default), pprof, dynaprof, mpip, gprof, psrun, hpm, packed, cube, hpc, ompp, snap, perixml, gptl, paraver, ipm, google
- i, --fixnames Use the fixnames option for gprof

## Notes

For the TAU profiles type, you can specify either a specific set of profile files on the commandline, or you can specify a directory (by default the current directory). The specified directory will be searched for profile.\*.\*.\* files, or, in the case of multiple counters, directories named MULTI\_\* containing profile data.

## Examples

```
perfdmf_loadtrial -e 12 -n "Batch 001"
```

This will load profile.\* (or multiple counters directories MULTI\_\*) into experiment 12 and give the trial the name "Batch 001"

```
perfdmf_loadtrial -e 12 -n "HPM data 01" -f hpm perfhpm*
```

This will load perfhpm\* files of type HPMToolkit into experiment 12 and give the trial the name "HPM data 01"

```
perfdmf_loadtrial -a "NPB2.3" -x "parametric" -n "64" par64.ppk
```

This will load packed profile par64.ppk into the experiment named "parametric" under the application named "NPB2.3" and give the trial the name "64". The application and experiment will be created if not found.

## Name

perfexplorer -- Launches TAU's Performance Data Mining Analyzer.

perfexplorer [-n, --ogui] [-i, --script *script*]

## Documentation

Complete documentation can be found at <http://www.cs.uoregon.edu/research/tau/tau-usersguide.pdf>

## Name

perfexplorer\_configure -- Configures a perfdfmf database for use with perfexplorer.

`perfexplorer_configure`

## Description

Configures a perfdfmf database for use with perfexplorer.

## Name

taucc -- C compiler wrapper for TAU

taucc [options] ...

## Options

-tau:help	Displays help
-tau:verbose	Enable verbose mode
-tau:keepfiles	Keep intermediate files
-tau:show	Do not invoke, just show what would be done
-tau:pdtinst	Use PDT instrumentation
-tau:compinst	Use compiler instrumentation
-tau:headerinst	Instrument headers
-tau:<options>	Specify measurement/instrumentation options. Sample options: mpi,pthread,openmp,profile,callpath,trace,vampirtrace,epilog
-tau:makefile tau_stub_makefile	Specify tau stub makefile

## Notes

If the -tau:makefile option is not used, the TAU\_MAKEFILE environment variable will be checked, if it is not specified, then the -tau:<options> will be used to identify a binding.

## Examples

taucc foo.c -o foo

taucc -tau:MPI,OPENMP,TRACE foo.c -o foo

taucc -tau:verbose -tau:PTHREAD foo.c -o foo

## Documentation

Complete documentation can be found at <http://www.cs.uoregon.edu/research/tau/tau-usersguide.pdf>

## Name

taucxx -- C++ compiler wrapper for TAU

taucxx [options] ...

## Options

-tau:help	Displays help
-tau:verbose	Enable verbose mode
-tau:keepfiles	Keep intermediate files
-tau:show	Do not invoke, just show what would be done
-tau:pdtinst	Use PDT instrumentation
-tau:compinst	Use compiler instrumentation
-tau:headerinst	Instrument headers
-tau:<options>	Specify measurement/instrumentation options. Sample options: mpi,pthread,openmp,profile,callpath,trace,vampirtrace,epilog
-tau:makefile <i>tau_stub_makefile</i>	Specify tau stub makefile

## Notes

If the -tau:makefile option is not used, the TAU\_MAKEFILE environment variable will be checked, if it is not specified, then the -tau:<options> will be used to identify a binding.

## Examples

taucxx foo.cpp -o foo

taucxx -tau:MPI,OPENMP,TRACE foo.cpp -o foo

taucxx -tau:verbose -tau:PTHREAD foo.cpp -o foo

## Documentation

Complete documentation can be found at <http://www.cs.uoregon.edu/research/tau/tau-usersguide.pdf>

## Name

tauf90 -- Fortran compiler wrapper for TAU

tauf90 [options] ...

## Options

-tau:help	Displays help
-tau:verbose	Enable verbose mode
-tau:keepfiles	Keep intermediate files
-tau:show	Do not invoke, just show what would be done
-tau:pdtinst	Use PDT instrumentation
-tau:compinst	Use compiler instrumentation
-tau:headerinst	Instrument headers
-tau:<options>	Specify measurement/instrumentation options. Sample options: mpi,pthread,openmp,profile,callpath,trace,vampirtrace,epilog
-tau:makefile tau_stub_makefile	Specify tau stub makefile

## Notes

If the -tau:makefile option is not used, the TAU\_MAKEFILE environment variable will be checked, if it is not specified, then the -tau:<options> will be used to identify a binding.

## Examples

tauf90 foo.f90 -o foo

tauf90 -tau:MPI,OPENMP,TRACE foo.f90 -o foo

tauf90 -tau:verbose -tau:PTHREAD foo.f90 -o foo

## Documentation

Complete documentation can be found at <http://www.cs.uoregon.edu/research/tau/tau-usersguide.pdf>

## Name

paraprof -- Launches TAU's Java-based performance data viewer.

paraprof [-h, --help] [-f, --filetype *filetype*] [--pack *file*] [--dump] [-o, --oss] [-s, --summary]

## Notes

For the TAU profiles type, you can specify either a specific set of profile files on the commandline, or you can specify a directory (by default the current directory). The specified directory will be searched for profile.\*.\*.\* files, or, in the case of multiple counters, directories named MULTL\_\* containing profile data.

## Options

-h	Display help
-f, --filetype <i>filetype</i>	Specify type of performance data. Options are: profiles (default), pprof, dynaprof, mpip, gprof, psrun, hpm, packed, cube, hpc, ompp, snap, perixml, gptl
--pack <i>file</i>	Pack the data into packed (.ppk) format (does not launch ParaProf GUI)
--dump	Dump profile data to TAU profile format (does not launch ParaProf GUI).
-o, --oss	Print profile data in OSS style text output
-s, --summary	Print only summary statistics (only applies to OSS output)

## Documentation

Complete documentation can be found at <http://www.cs.uoregon.edu/research/tau/tau-usersguide.pdf>

## Name

pprof -- Quickly displays profile data.

pprof [-a] [-c] [-b] [-m] [-t] [-e] [-i] [-v] [-r] [-s] [-n num] [-f filename] [-p] [-l] [-d]

## Description

## Options

- a Show all location information available
- c Sort according to number of Calls
- b Sort according to number of subroutines called by a function
- m Sort according to Milliseconds (exclusive time total)
- t Sort according to Total milliseconds (inclusive time total) (default)
- e Sort according to Exclusive time per call (msec/call)
- i Sort according to Inclusive time per call (total msec/call)
- v Sort according to Standard Deviation (excl usec)
- r Reverse sorting order
- s print only Summary profile information
- n num print only first num number of functions
- f filename specify full path and filename without node ids
- p suPpress conversion to hhmmssmmm format
- l List all functions and exit
- d Dump output format (for tau\_reduce) [node numbers] prints only info about all contexts/threads of given node numbers

## Name

tau\_instrumentor -- automatically instruments a source basied on information provided by pdt.  
tau\_instrumentor [--help] {*pdbfile*} {*sourcefile*} [-c] [-b] [-m] [-t] [-e] [-i] [-v] [-r] [-s]  
[-n *num*] [-f *filename*] [-p] [-l] [-d]

## Description

## Options

- a Show all location information available
- c Sort according to number of Calls
- b Sort according to number of suBroutines called by a function
- m Sort according to Milliseconds (exclusive time total)
- t Sort according to Total milliseconds (inclusive time total) (default)
- e Sort according to Exclusive time per call (msec/call)
- i Sort according to Inclusive time per call (total msec/call)
- v Sort according to Standard Deviation (excl usec)
- r Reverse sorting order
- s print only Summary profile information
- n *num* print only first *num* number of functions
- f *filename* specify full path and Filename without node ids
- p suPpress conversion to hhmmssmmm format
- l List all functions and exit
- d Dump output format (for tau\_reduce) [node numbers] prints only info about all contexts/threads of given node numbers

## Example

```
%> tau_instrumentor foo.pdb foo.cpp -o foo.inst.cpp -f select.tau
```

## Name

```
vtfconverter --  
vtfconverter [-h] [-c] [-f file] [-p path] [-i from to]
```

## Description

Converts VTF profile to TAU profiles and launches an interactive VTF prompt.

## Options

- c Opens command line interface.
- f Converts trace [file] to TAU profiles.
- p Places the resulting profiles in the directory [path].
- i States that the interval [from],[to] should be profiled.

## Name

tau\_setup -- Launches GUI interface to configure TAU.

`tau_setup`

## Options

`-v` Verbose output.

`--html` Output results in HTML.

`--build` Only build.

`--run` Only run.

## Name

tau\_wrap -- Instruments an external library with TAU without needing to recompile

```
tau_wrap {pdbfile} {sourcefile} [-o outputfile] [-g groupname] [-i headerfile] [-f selectivefile]
```

## Options

pdbfile	A pdb file generated by cparse, cxxparse, or f90parse; these commands are found in the [PDT_HOME]/[arch]/bin directory.
sourcefile	The source file corresponding to the pdbfile.
-o outputfile	The filename of the resulting instrumented source file.
-g groupname	This associates all the functions profiled as belonging to the this group. Once profiled you will be able to analysis these functions separately.
-i headerfile	By default tau_wrap will include Profile/Profile.h; use this option to specify a different header file.
-f selectivefile	You can specify a selective instrumentation file that defines how the source file is to be instrumented.

## Examples

```
%> tau_wrap hdf5.h.pdb hdf5.h -o hdf5.inst.c -f select.tau -g hdf5
```

This specifies the instrumented wrapper library source (hdf5.inst.c), the instrumentation specification file (select.tau) and the group (hdf5). It creates the wrapper/ directory.

## Name

tau\_gen\_wrapper -- Generates a wrapper library that can intercept at link time or at runtime routines specified in a header file

```
tau_gen_wrapper {headerfile} {library} [-w | -d | -r ]
```

## Options

headerfile	Name of the headerfile to be wrapped
library	Name of the library to wrap
-w	(default) generates wrappers for re-linking the application
-d	generates wrappers by redefining routines during compilation in header files
-r	generates wrappers that may be pre-loaded using tau_exec at runtime

## Examples

```
%> tau_gen_wrapper hdf5.h /usr/lib/libhdf5.a
```

This generates a wrapper library that may be linked in using TAU\_OPTIONS -optTauWrapFile=<wrapperdir>/link\_options.tau

## Notes

tau\_gen\_wrapper reads the TAU\_MAKEFILE environment variable to get PDT settings

## Name

`tau_pin` -- Instruments application at run time using Intel's PIN library

`tau_pin [-n proc_num] [-r rules] {--} [myapp] [myargs]`

## Options

<code>-n proc_num</code>	This argument enables multiple instances of MPI applications launched with MPIEXEC. <code>proc_num</code> is the parameter indicating number of MPI process instances to be launched. This argument is optional and one can profile MPI application even with single process instance without this argument.
<code>-r rule</code>	This argument is specification rule for profiling the application. It allows selective profiling by specifying the "rule". The rule is a wildcard expression token which will indicate the area of profiling. It can be only the routine specification like "*" which indicates it'll instrument all the routines in the EXE or MPI routines. One can further specify the routines on a particular dll by the rule "somedll.dll!*". The dll name can also be in regular expression. We treat the application exe and MPI routines as special cases and specifying only the routines is allowed.
<code>myapp</code>	It's the application exe. This application can be Windows or console application. Profiling large Windows applications might suffer from degraded performance and interoperability. Specifying a limited number of interesting routines can help.
<code>myargs</code>	It's the command line arguments of the application.

## Examples

To profile routines in mytest.exe with prefix "myf":

```
tau_pin -r myf.* -- mytest.exe
```

To profile all routines in mpitest.exe ( no need to specify any rule for all ):

```
tau_pin mpitest.exe
```

to profile only MPI routines in mpitest.exe by launching two instances:

```
tau_pin -n 2 -r _MPI_.* -- mpitest.exe
```

## Wildcards

- \* for anything, for example \*MPI\* means any string having MPI in between any other characters.
- ? It's a placeholder wild card ?MPI\* means any character followed by MPI and followed by any string, example: ??Try could be \_\_Try or MyTry or MeTry etc.

## Name

tau\_java -- Instruments java applications at runtime using JVMTI

tau\_java [*optionsjavaprogram*] [*args*]

## Options

<i>-help</i>	Displays help information.
<i>-verbose</i>	Report the arguments of the script before it runs.
<i>-tau:agentlib=&lt;agentlib&gt;</i>	By default tau_java uses the most recently configured jdk, you can specify a different one here.
<i>-tau:java=&lt;javapath&gt;</i>	Path to a java binary, by default uses the one corresponding to the most recently configured jdk.
<i>-tau:bootclasspath=&lt;bootclasspath&gt;</i>	To modify the bootclasspath to point to a different jar, not usually necessary.
<i>-tau:include=&lt;item&gt;</i>	Only instrument these methods or classes. Separate multiple classes and methods with semicolons
<i>-tau:exclude=&lt;item&gt;</i>	Exclude the listed classes and methods. Separate multiple classes and methods with semicolons
<i>args</i>	the command line arguments of the java application.

## Name

`tau_cupti_avail` -- Detects the available CUPTI counters on the a each GPU device.

`tau_cupti_avail [-c counter names]`

## Options

`-c counter names` Checks which of a colon seperated list of CUPTI counter names can be recorded.

## Name

tau\_run -- Instruments and executes binaries to generate performance data. (DyninstAPI based instrumentor)

## Options

-v	optional verbose option
-o <i>outfile</i>	for binary rewriting
-T<option>	: specify TAU option
-loadlib=<file.so >	: specify additional load library
-XrunTAU-<options>	specify TAU library directly

## Name

tau\_rewrite -- Rewrites binaries using Maqao if Tau is configured using PDT 3.17+ at the routine level. If it doesn't find the Maqao package from PDT 3.17, it reverts to tau\_run (DyninstAPI based instrumentor).

## Options

<code>-o <i>outfile</i></code>	specify instrumented output file
<code>-T</code>	specify TAU option (CUPTI, DISABLE, MPI, OPENMP, PDT, PGI, PROFILE, SCOREP, SERIAL)
<code>-loadlib= <i>file.so</i></code>	specify additional load library
<code>-s</code>	dryrun without executing
<code>-v</code>	long verbose mode
<code>-v1</code>	short verbose mode
<code>-XrunTAUsh- <i>options</i></code>	specify TAU library directly

## Notes

Defaults if unspecified: -T MPI

MPI is assumed unless SERIAL is specified

## Example

```
tau_rewrite -T papi,pdt a.out -o a.inst
```

```
mpirun -np 4 ./a.inst
```

# **PerfDMF**

**University of Oregon**

---

# **PerfDMF**

by University of Oregon

Published (TBA)

Copyright © 2005 University of Oregon Performance Research Lab

---

---

---

# Table of Contents

1. Introduction .....	5
1.1. Prerequisites .....	5
1.2. Installation .....	5
2. Using PerfDMF .....	7
2.1. perfdmf_createapp .....	7
2.2. perfdmf_createexp .....	7
2.3. perfdmf_loadtrial .....	7

---

# Chapter 1. Introduction

PerfDMF (Performance Data Management Framework) is a an API/Toolkit that sits atop a DBMS to manage and analyze performance data. The API is available in its native Java form as well as C.

## 1.1. Prerequisites

1. A supported database (currently, PostgreSQL, MySQL, or Oracle).
2. Java 1.4.

## 1.2. Installation

The PerfDMF utilities and applications are installed as part of the standard TAU release. Shell scripts are installed in the TAU bin directory to configure and run the utilities. It is assumed that the user has installed TAU and run TAU's configure and 'make install'.

1. Create a database. This step will depend on the user's chosen database.

- **PostgreSQL:**

```
$ createdb -O perfdfmf perfdfmf
```

Or, from **psql**

```
psql=# create database perfdfmf with owner = perfdfmf;
```

- **MySQL:** From the MySQL prompt

```
mysql> create database perfdfmf;
```

- **Oracle:** It is recommended that you create a tablespace for perfdfmf:

```
create tablespace perfdfmf  
datafile '/path/to/somewhere' size 500m reuse;
```

Then, create a user that has this tablespace as default:

```
create user amorris identified by db;  
grant create session to amorris;  
grant create table to amorris;  
grant create sequence to amorris;  
grant create trigger to amorris;  
alter user amorris quota unlimited on perfdfmf;  
alter user amorris default tablespace perfdfmf;
```

PerfDMF is set up to use the Oracle Thin Java driver. You will have to obtain this jar file for your database. In our case, it was ojdbc14.jar

2. Configure PerfDMF. To configure PerfDMF, run the **perfdfmf\_configure** from the TAU bin directory.

The configuration program will prompt the user for several values. The default values will work for most users. When configuration is complete, it will connect to the database and test the configuration. If the configuration is valid and the schema is not found (as will be the case on initial configuration), the schema will be uploaded. Be sure to specify the correct schema for your database.

---

# Chapter 2. Using PerfDMF

The easiest way to interact with PerfDMF is to use ParaProf which provides a GUI interface to all of the database information. In addition, the following commandline utilities are provided.

## 2.1. perfdfmf\_createapp

This utility creates applications with a given name

```
$ perfdfmf_createapp -n "New Application"
Created Application, ID: 24
```

## 2.2. perfdfmf\_createexp

This utility creates experiments with a given name, under a specified application

```
$ perfdfmf_createexp -a 24 -n "New Experiment"
Created Experiment, ID: 38
```

## 2.3. perfdfmf\_loadtrial

This utility uploads a trial to the database with a given name, under a specified experiment

```
Usage: perfdfmf_loadtrial -e <experiment id> -n <name>
[options] <files>
```

Required Arguments:

-e, --experimentid <number>	Specify associated experiment ID for the trial
-n, --name <text>	Specify the name of the trial

Optional Arguments:

-f, --filetype <filetype>	Specify type of performance data, options are: profiles (default), pprof, dynaprof, mpip, gprof, psrun, hpm, packed, cube, hpc
-t, --trialid <number>	Specify trial ID
-i, --fixnames	Use the fixnames option for gprof

Notes:

For the TAU profiles type, you can specify either a specific set of profile files on the commandline, or you can specify a directory (by default the current directory). The specified directory will be searched for profile.\*.\*.\* files, or, in the case of multiple counters, directories named MULTI\_\* containing profile data.

Examples:

```
perfdmf_loadtrial -e 12 -n "Batch 001"
This will load profile.* (or multiple counters directories MULTI_*)
into experiment 12 and give the trial the name "Batch 001"

perfdmf_loadtrial -e 12 -n "HPM data 01" perfhpm*
```

This will load perfhpm\* files of type HPMToolkit into experiment  
12 and give the trial the name "HPM data 01"

---

# Chapter 3. Windows

## 3.1. TAU on Windows

### 3.1.1. Installation

We provide a binary release build for Windows on the download page [<http://www.cs.uoregon.edu/research/tau/downloads.php>]. TAU can also be built from source using `Makefile.win32`.

### 3.1.2. Instrumenting an application with Visual Studio C/C++

Here is a step by step guide for retrieving a standard profile from a threaded program.

1. Download TAU (see previous section)
2. Open [TAU-HOME]/examples/threads/threads.sln in VC 7 or greater.
3. Open `testTau.cpp` source file.
4. Uncomment the pragma element at the top of the file so that it reads:

```
#define PROFILING_ON 1
#pragma comment(lib, "tau-profile-static-mt.lib")
```
5. Edit these properties of this project:
  - a. Add the ..\..\lib\vc7\ directory to the Linker's Additional Library Directories.
  - b. Set the Runtime Library to Multi-threaded DLL (MD) in the C/C++ Code Generation section.
6. Build and run the application.
7. Launch Visual Studio's command line prompt. Move to the [TAU-HOME]/examples/threads/directory/ this is where the profile files were written. Type:

```
%> [TAU-HOME]/bin/paraprof
```

To view these profiles in paraprof

### 3.1.3. Using MINGW with TAU

Building TAU with the MinGW cross-compilers for 32- or 64-bit Windows Requirements:

MinGW compilers must be in your path. For example (64-bit):

- \* x86\_64-w64-mingw32-gcc
- \* x86\_64-w64-mingw32-g++
- \* x86\_64-w64-mingw32-ar
- \* x86\_64-w64-mingw32-ld
- \* x86\_64-w64-mingw32-ranlib

Limitations:

- \* No signal processing
- \* No event-based sampling (EBS)

Instructions:

See ./configure -help.

---

# TAU Instrumentation API

## Introduction

- **C++**

The C++ API is a set of macros that can be inserted in the C++ source code. An extension of the same API is available to instrument C and Fortran sources.

At the beginning of each instrumented source file, include the following header

```
#include <TAU.h>
```

- **C**

The API for instrumenting C source code is similar to the C++ API. The primary difference is that the TAU\_PROFILE() macro is not available for identifying an entire block of code or function. Instead, routine transitions are explicitly specified using TAU\_PROFILE\_TIMER() macro with TAU\_PROFILE\_START() and TAU\_PROFILE\_STOP() macros to indicate the entry and exit from a routine. Note that, TAU\_TYPE\_STRING() and CT() macros are not applicable for C. It is important to declare the TAU\_PROFILE\_TIMER() macro after all the variables have been declared in the function and before the execution of the first C statement.

Example:

```
#include <TAU.h>

int main (int argc, char **argv) {
    int ret;
    pthread_attr_t attr;
    pthread_t tid;
    TAU_PROFILE_TIMER(tautimer,"main()", "int (int, char **)",
                      TAU_DEFAULT);
    TAU_PROFILE_START(tautimer);
    TAU_PROFILE_INIT(argc, argv);
    TAU_PROFILE_SET_NODE(0);
    pthread_attr_init(&attr);
    printf("Started Main...\n");
    // other statements
    TAU_PROFILE_STOP(tautimer);
    return 0;
}
```

- **Fortran 77/90/95**

The Fortran90 TAU API allows source code written in Fortran to be instrumented for TAU. This API is comprised of Fortran routines. As explained in Chapter 2, the instrumentation can be disabled in the program by using the TAU stub makefile variable TAU\_DISABLE on the link command line. This points to a library that contains empty TAU instrumentation routines.

# Timers

- **Static timers**

These are commonly used in most profilers where all invocations of a routine are recorded. The name and group registration takes place when the timer is created (typically the first time a routine is entered). A given timer is started and stopped at routine entry and exit points. A user defined timer can also measure the time spent in a group of statements. Timers may be nested but they may not overlap. The performance data generated can typically answer questions such as: *what is the total time spent in MPI\_Send() across all invocations?*

- **Dynamic timers**

To record the execution of each invocation of a routine, TAU provides dynamic timers where a unique name may be constructed for a dynamic timer for each iteration by embedding the iteration count in it. It uses the start/stop calls around the code to be examined, similar to static timers. The performance data generated can typically answer questions such as: *what is the time spent in the routine foo() in iterations 24, 25, and 40?*

- **Static phases**

An application typically goes through several phases in its execution. To track the performance of the application based on phases, TAU provides static and dynamic phase profiling. A profile based on phases highlights the context in which a routine is called. An application has a default phase within which other routines and phases are invoked. A phase based profile shows the time spent in a routine when it was in a given phase. So, if a set of instrumented routines are called directly or indirectly by a phase, we'd see the time spent in each of those routines under the given phase. Since phases may be nested, a routine may belong to only one phase. When more than one phase is active for a given routine, the closest ancestor phase of a routine along its callstack is its phase for that invocation. The performance data generated can answer questions such as: *what is the total time spent in MPI\_Send() when it was invoked in all invocations of the IO (IO => MPI\_Send()) phase?*

- **Dynamic phases**

Dynamic phases borrow from dynamic timers and static phases to create performance data for all routines that are invoked in a given invocation of a phase. If we instrument a routine as a dynamic phase, creating a unique name for each of its invocations (by embedding the invocation count in the name), we can examine the time spent in all routines and child phases invoked directly or indirectly from the given phase. The performance data generated can typically answer questions such as: *what is the total time spent in MPI\_Send() when it was invoked directly or indirectly in iteration 24?* Dynamic phases are useful for tracking per-iteration profiles for an adaptive computation where iterations may differ in their execution times.

- **Callpaths**

In phase-based profiles, we see the relationship between routines and parent phases. Phase profiles do not show the calling structure between different routines as is represented in a callgraph. To do so, TAU provides callpath profiling capabilities where the time spent in a routine along an edge of a callgraph is captured. Callpath profiles present the full flat profiles of routines (or nodes in the callgraph), as well as routines along a callpath. A callpath is represented syntactically as a list of routines separated by a delimiter. The maximum depth of a callpath is controlled by an environment variable.

- **User-defined Events**

Besides timers and phases that measure the time spent between a pair of start and stop calls in the code, TAU also provides support for user-defined atomic events. After an event is registered with a

name, it may be triggered with a value at a given point in the source code. At the application level, we can use user-defined events to track the progress of the simulation by keeping track of application specific parameters that explain program dynamics, for example, the number of iterations required for convergence of a solver at each time step, or the number of cells in each iteration of an adaptive mesh refinement application.

---

## Name

TAU\_START -- Starts a timer.

C/C++:

```
TAU_START(name);
char* name;
```

Fortran:

```
TAU_START(name);
character name(2);
```

## Description

Starts the timer given by *name*

## Example

C/C++ :

```
int foo(int a) {
    TAU_START("t1");
    ...
    TAU_STOP("t2");
    return a;
}
```

Fortran :

```
subroutine F1()
    character(13) cvar

    write (cvar,'(a9,i2)') 'Iteration', val
        call TAU_START(cvar)
    ...
    call TAU_STOP(cvar)
end
```

## See Also

TAU\_PROFILE, TAU\_STOP

---

## Name

TAU\_STOP -- Stops a timer.

C/C++:

```
TAU_STOP(name);  
char* name;
```

Fortran:

```
TAU_STOP(name);  
character name(2);
```

## Description

Stops the timer given by *timer*. It is important to note that timers can be nested, but not overlapping. TAU detects programming errors that lead to such overlaps at runtime, and prints a warning message.

## Example

C/C++ :

```
int foo(int a) {  
    TAU_START("t1");  
    ...  
    TAU_STOP("t2");  
    return a;  
}
```

Fortran :

```
subroutine F1()  
    character(13) cvar  
  
    write (cvar,'(a9,i2)') 'Iteration', val  
    call TAU_START(cvar)  
    ...  
    call TAU_STOP(cvar)  
end
```

## See Also

TAU\_PROFILE, TAU\_START

---

## Name

TAU\_PROFILE -- Profile a C++ function

```
TAU_PROFILE(function_name, type, group);  
char* or string& function_name;  
char* or string& type;  
TauGroup_t group;
```

## Description

TAU\_PROFILE profiles a function. This macro defines the function and takes care of the timer start and stop as well. The timer will stop when the macro goes out of scope (as in C++ destruction).

## Example

```
int foo(char *str) {  
    TAU_PROFILE(foo, "int (char *)", TAU_DEFAULT);  
    ...  
}
```

## See Also

TAU\_PROFILE\_TIMER

---

## Name

TAU\_DYNAMIC\_PROFILE -- dynamic\_profile a c++ function

```
TAU_DYNAMIC_PROFILE(function_name, type, group);
char* or string& function_name;
char* or string& type;
taugroup_t group;
```

## description

TAU\_DYNAMIC\_PROFILE profiles a function dynamically creating a separate profile for each time the function is called. this macro defines the function and takes care of the timer start and stop as well. the timer will stop when the macro goes out of scope (as in c++ destruction).

## example

```
int foo(char *str) {
    tau_dynamic_profile("foo", "int (char *)", tau_default);
    ...
}
```

---

## Name

TAU\_PROFILE\_CREATE\_DYNAMIC -- Creates a dynamic timer

C/C++:

```
TAU_PROFILE_CREATE_DYNAMIC(timer, function_name, type, group);
Timer timer;
char* or string& function_name;
char* or string& type;
taugroup_t group;
```

Fortran:

```
TAU_PROFILE_CREATE_DYNAMIC(timer, name);
integer timer(2);
character name(size);
```

## description

TAU\_PROFILE\_CREATE\_DYNAMIC creates a dynamic timer the name of the timer should be different for each execution.

## example

>C/C++:

```
int main(int argc, char **argv) {
    int i;
    TAU_PROFILE_TIMER(t, "main()", "", TAU_DEFAULT);
    TAU_PROFILE_SET_NODE(0);
    TAU_PROFILE_START(t);

    for (i=0; i<5; i++) {
        char buf[32];
        sprintf(buf, "Iteration %d", i);

        TAU_PROFILE_CREATE_DYNAMIC(timer, buf, "", TAU_USER);
        TAU_PROFILE_START(timer);
        printf("Iteration %d\n", i);
        f1();

        TAU_PROFILE_STOP(timer);
    }
    return 0;
}
```

>Fortran:

```
subroutine ITERATION(val)
    integer val
    character(13) cvar
    integer profiler(2) / 0, 0 /
    save profiler
```

```
print *, "Iteration ", val
write (cvar,'(a9,i2)') 'Iteration', val
call TAU_PROFILE_CREATE_DYNAMIC(profiler, cvar)
call TAU_PROFILE_START(profiler)

call F1()
call TAU_PROFILE_STOP(profiler)
return
end
```

## see also

[TAU\\_DYNAMIC\\_TIMER\\_START](#)

[TAU\\_DYNAMIC\\_TIMER\\_STOP](#)

---

## Name

TAU\_CREATE\_DYNAMIC\_AUTO -- Creates a dynamic timer for C/C++

```
TAU_CREATE_DYNAMIC_AUTO(timer, function_name, type, group);  
Timer timer;  
char* or string& function_name;  
char* or string& type;  
taugroup_t group;
```

## description

TAU\_CREATE\_DYNAMIC\_AUTO creates a dynamic timer automatically incrementing the name each time the timer is executed.

## example

```
int tau_ret_val;  
TAU_PROFILE_CREATE_DYNAMIC_AUTO(tautimer, "int fool(int) C [{foo.c} {22,1}-{29,1}]"  
TAU_PROFILE_START(tautimer);  
{  
printf("inside fool: calling bar: x = %d\n", x);  
printf("before calling bar in fool\n");  
bar(x-1); /* 26 */  
printf("after calling bar in fool\n");  
{ tau_ret_val = x; TAU_PROFILE_STOP(tautimer); return (tau_ret_val); }
```

## see also

[TAU\\_PROFILE\\_CREATE\\_DYNAMIC](#)

[TAU\\_DYNAMIC\\_TIMER\\_START](#)

[TAU\\_DYNAMIC\\_TIMER\\_STOP](#)

---

## Name

TAU\_PROFILE\_DYNAMIC\_ITER -- Creates a dynamic timer in Fortran.

```
TAU_PROFILE_DYNAMIC_ITER(iterator, timer, name);
integer iterator;
integer timer(2);
character name(size);
```

## description

TAU\_PROFILE\_DYNAMIC\_ITER creates a dynamic timer the name of the timer is appended by the iterator.

## example

```
integer tau_iter / 0 /
save tau_iter
tau_iter = tau_iter + 1
call TAU_PROFILE_DYNAMIC_ITER(tau_iter, profiler, '
&FOO1 [{foo.f90} {16,18}]')
call TAU_PROFILE_START(profiler)
print *, "inside fool: calling bar, x = ", x
call bar(x-1)
print *, "after calling bar"
call TAU_PROFILE_STOP(profiler)
```

## see also

[TAU\\_DYNAMIC\\_TIMER\\_START](#)

[TAU\\_DYNAMIC\\_TIMER\\_STOP](#)

---

## Name

TAU\_PHASE\_DYNAMIC\_ITER -- Creates a dynamic phase in Fortran.

```
TAU_PHASE_DYNAMIC_ITER(iterator, timer, name);
integer iterator;
integer timer(2);
character name(size);
```

## description

TAU\_PHASE\_DYNAMIC\_ITER creates a dynamic phase the name of which is appended by the iterator.

## example

```
      integer tau_iter / 0 /
save tau_iter
tau_iter = tau_iter + 1
call TAU_PHASE_DYNAMIC_ITER(tau_iter, profiler,
&FOO1 [{foo.f90} {16,18}])
call TAU_PHASE_START(profiler)
print *, "inside fool: calling bar, x = ", x
call bar(x-1)
print *, "after calling bar"
call TAU_PROFILE_STOP(profiler)
```

## see also

TAU\_DYNAMIC\_TIMER\_START

TAU\_DYNAMIC\_TIMER\_STOP

---

## Name

TAU\_PROFILE\_TIMER -- Defines a static timer.

C/C++:

```
TAU_PROFILE_TIMER(timer, function_name, type, group);  
Profiler timer;  
char* or string& function_name;  
char* or string& type;  
TauGroup_t group;
```

Fortran:

```
TAU_PROFILE_TIMER(profiler, name);  
integer profiler(2);  
character name(size);
```

## Description

C/C++ :

With TAU\_PROFILE\_TIMER, a group of one or more statements is profiled. This macro has a timer variable as its first argument, and then strings for name and type, as described earlier. It associates the timer to the profile group specified in the last parameter.

Fortran :

To profile a block of Fortran code, such as a function, subroutine, loop etc., the user must first declare a profiler, which is an integer array of two elements (pointer) with the save attribute, and pass it as the first parameter to the TAU\_PROFILE\_TIMER subroutine. The second parameter must contain the name of the routine, which is enclosed in a single quote. TAU\_PROFILE\_TIMER declares the profiler that must be used to profile a block of code. The profiler is used to profile the statements using TAU\_PROFILE\_START and TAU\_PROFILE\_STOP as explained later.

## Example

C/C++ :

```
template< class T, unsigned Dim >  
void BareField<T,Dim>::fillGuardCells(bool reallyFill)  
{  
    // profiling macros  
    TAU_TYPE_STRING(taustr, CT(*this) + " void (bool)" );  
    TAU_PROFILE("BareField::fillGuardCells()", taustr, TAU_FIELD);  
    TAU_PROFILE_TIMER(sendtimer, "fillGuardCells-send",  
                      taustr, TAU_FIELD);  
    TAU_PROFILE_TIMER(localstimer, "fillGuardCells-locals",  
                      taustr, TAU_FIELD);  
    ...  
}
```

Fortran :

```
subroutine bcast_inputs
implicit none
integer profiler(2)
save profiler

include 'mpinpb.h'
include 'applu.incl'

interger IERR

call TAU_PROFILE_TIMER(profiler, 'bcast_inputs')
```

## See Also

TAU\_PROFILE\_TIMER\_DYNAMIC, TAU\_PROFILE\_START, TAU\_PROFILE\_STOP

---

## Name

TAU\_PROFILE\_START -- Starts a timer.

C/C++:

```
TAU_PROFILE_START(timer);
Profiler timer;
```

Fortran:

```
TAU_PROFILE_START(profiler);
integer profiler(2);
```

## Description

Starts the timer given by *timer*

## Example

C/C++ :

```
int foo(int a) {
    TAU_PROFILE_TIMER(timer, "foo", "int (int)", TAU_USER);
    TAU_PROFILE_START(timer);
    ...
    TAU_PROFILE_STOP(timer);
    return a;
}
```

Fortran :

```
subroutine f1()
    integer profiler(2) / 0, 0 /
    save    profiler

    call TAU_PROFILE_TIMER(profiler,'f1()')
    call TAU_PROFILE_START(profiler)
    ...
    call TAU_PROFILE_STOP(profiler)
end
```

## See Also

TAU\_PROFILE\_TIMER, TAU\_PROFILE\_STOP

---

## Name

TAU\_PROFILE\_STOP -- Stops a timer.

C/C++:

```
TAU_PROFILE_STOP(timer);
Profiler timer;
```

Fortran:

```
TAU_PROFILE_STOP(profiler);
integer profiler(2);
```

## Description

Stops the timer given by *timer*. It is important to note that timers can be nested, but not overlapping. TAU detects programming errors that lead to such overlaps at runtime, and prints a warning message.

## Example

C/C++ :

```
int foo(int a) {
    TAU_PROFILE_TIMER(timer, "foo", "int (int)", TAU_USER);
    TAU_PROFILE_START(timer);
    ...
    TAU_PROFILE_STOP(timer);
    return a;
}
```

Fortran :

```
subroutine f1()
    integer profiler(2) / 0, 0 /
    save    profiler

    call TAU_PROFILE_TIMER(profiler,'f1()')
    call TAU_PROFILE_START(profiler)
    ...
    call TAU_PROFILE_STOP(profiler)
end
```

## See Also

TAU\_PROFILE\_TIMER, TAU\_PROFILE\_START

---

## Name

TAU\_STATIC\_TIMER\_START -- Starts a timer.

C/C++:

```
TAU_STATIC_TIMER_START(timer);
Profiler timer;
```

Fortran:

```
TAU_STATIC_TIMER_START(profiler);
integer profiler(2);
```

## Description

Starts a static timer defined by TAU\_PROFILE.

## Example

C/C++ :

```
TAU_PROFILE("int foo(int) [{foo.cpp} {13,1}-{20,1}]", " ", TAU_USER);

printf("inside foo: calling bar: x = %d\n", x);
printf("before calling bar in foo\n");
TAU_STATIC_TIMER_START("foo_bar");
bar(x-1); /* 17 */
printf("after calling bar in foo\n");
TAU_STATIC_TIMER_STOP("foo_bar");
```

Fortran :

```
call TAU_PROFILE_TIMER(profiler, 'FOO [{foo.f90} {8,18}]')
call TAU_PROFILE_START(profiler)
print *, "inside foo: calling bar, x = ", x
  call TAU_STATIC_TIMER_START("foo_bar");
    call bar(x-1)
  print *, "after calling bar"
    call TAU_STATIC_TIMER_STOP("foo_bar");
call TAU_PROFILE_STOP(profiler)
```

## See Also

TAU\_PROFILE, TAU\_STATIC\_PHASE\_START, TAU\_STATIC\_PHASE\_STOP

---

## Name

TAU\_STATIC\_TIMER\_STOP -- Starts a timer.

C/C++:

```
TAU_STATIC_TIMER_STOP(timer);
Profiler timer;
```

Fortran:

```
TAU_STATIC_TIMER_STOP(profiler);
integer profiler(2);
```

## Description

Starts a static timer defined by TAU\_PROFILE.

## Example

C/C++ :

```
TAU_PROFILE("int foo(int) [{foo.cpp} {13,1}-{20,1}]", " ", TAU_USER);

printf("inside foo: calling bar: x = %d\n", x);
printf("before calling bar in foo\n");
TAU_STATIC_TIMER_START("foo_bar");
bar(x-1); /* 17 */
printf("after calling bar in foo\n");
TAU_STATIC_TIMER_STOP("foo_bar");
```

Fortran :

```
call TAU_PROFILE_TIMER(profiler, 'FOO [{foo.f90} {8,18}]')
call TAU_PROFILE_START(profiler)
print *, "inside foo: calling bar, x = ", x
  call TAU_STATIC_TIMER_START("foo_bar");
    call bar(x-1)
  print *, "after calling bar"
    call TAU_STATIC_TIMER_STOP("foo_bar");
call TAU_PROFILE_STOP(profiler)
```

## See Also

TAU\_PROFILE, TAU\_STATIC\_PHASE\_START, TAU\_STATIC\_PHASE\_STOP

---

## Name

TAU\_DYNAMIC\_TIMER\_START -- Starts a dynamic timer.

C/C++:

```
TAU_DYNAMIC_TIMER_START(name);
String name;
```

Fortran:

```
TAU_DYNAMIC_TIMER_START(iteration, name);
integer iteration;
char name(size);
```

## Description

Starts a new dynamic timer concating the iterator to the end of the name.

## Example

C/C++ :

```
int foo(int a) {
    TAU_PROFILE_TIMER(timer, "foo", "int (int)", TAU_USER);
    TAU_DYNAMIC_TIMER_START(timer);
    ...
    TAU_PROFILE_STOP(timer);
    return a;
}
```

Fortran :

```
integer tau_iteration / 0 /
save tau_iteration
call TAU_PROFILE_TIMER(profiler, 'FOO1 [{foo.f90} {16,18}]')
call TAU_PROFILE_START(profiler)
print *, "inside fool: calling bar, x = ", x
tau_iteration = tau_iteration + 1
call TAU_DYNAMIC_TIMER_START(tau_iteration,"fool_bar");
    call bar(x-1)
print *, "after calling bar"
    call TAU_DYNAMIC_TIMER_STOP(tau_iteration,"fool_bar");
call TAU_PROFILE_STOP(profiler)
```

## See Also

TAU\_PROFILE\_TIMER, TAU\_PROFILE\_STOP

---

## Name

TAU\_DYNAMIC\_TIMER\_STOP -- Starts a dynamic timer.

C/C++:

```
TAU_DYNAMIC_TIMER_STOP(name);
String name;
```

Fortran:

```
TAU_DYNAMIC_TIMER_STOP(iteration, name);
integer iteration;
char name(size);
```

## Description

Stops a new dynamic timer concating the iterator to the end of the name.*timer*

## Example

C/C++ :

```
int foo(int a) {
    TAU_PROFILE_TIMER(timer, "foo", "int (int)", TAU_USER);
    TAU_DYNAMIC_TIMER_START(timer);
    ...
    TAU_PROFILE_STOP(timer);
    return a;
}
```

Fortran :

```
integer tau_iteration / 0 /
save tau_iteration
call TAU_PROFILE_TIMER(profiler, 'FOO1 [{foo.f90} {16,18}]')
call TAU_PROFILE_START(profiler)
print *, "inside foo: calling bar, x = ", x
tau_iteration = tau_iteration + 1
call TAU_DYNAMIC_TIMER_START(tau_iteration,"fool_bar");
    call bar(x-1)
print *, "after calling bar"
    call TAU_DYNAMIC_TIMER_STOP(tau_iteration,"fool_bar");
call TAU_PROFILE_STOP(profiler)
```

## See Also

TAU\_PROFILE\_TIMER, TAU\_PROFILE\_STOP

---

## Name

TAU\_PROFILE\_TIMER\_DYNAMIC -- Defines a dynamic timer.

C/C++:

```
TAU_PROFILE_TIMER_DYNAMIC(timer, function_name, type, group);
Profiler timer;
char* or string& function_name;
char* or string& type;
TauGroup_t group;
```

Fortran:

```
TAU_PROFILE_TIMER_DYNAMIC(profiler, name);
integer profiler(2);
character name(size);
```

## Description

TAU\_PROFILE\_TIMER\_DYNAMIC operates similar to TAU\_PROFILE\_TIMER except that the timer is created each time the statement is invoked. This way, the name of the timer can be different for each execution.

## Example

C/C++ :

```
int main(int argc, char **argv) {
    int i;
    TAU_PROFILE_TIMER(t, "main()", "", TAU_DEFAULT);
    TAU_PROFILE_SET_NODE(0);
    TAU_PROFILE_START(t);

    for (i=0; i<5; i++) {
        char buf[32];
        sprintf(buf, "Iteration %d", i);

        TAU_PROFILE_TIMER_DYNAMIC(timer, buf, "", TAU_USER);
        TAU_PROFILE_START(timer);
        printf("Iteration %d\n", i);
        f1();

        TAU_PROFILE_STOP(timer);
    }
    return 0;
}
```

Fortran :

```
subroutine ITERATION(val)
    integer val
    character(13) cvar
    integer profiler(2) / 0, 0 /
```

```
save profiler
print *, "Iteration ", val
write (cvar,'(a9,i2)') 'Iteration', val
call TAU_PROFILE_TIMER_DYNAMIC(profiler, cvar)
call TAU_PROFILE_START(profiler)

call F1()
call TAU_PROFILE_STOP(profiler)
return
end
```

## See Also

[TAU\\_PROFILE\\_TIMER](#), [TAU\\_PROFILE\\_START](#), [TAU\\_PROFILE\\_STOP](#)

---

## Name

TAU\_PROFILE\_DECLARE\_TIMER -- Declares a timer for C

C:

```
TAU_PROFILE_DECLARE_TIMER(timer);
Profiler timer;
```

## Description

Because C89 does not allow mixed code and declarations, TAU\_PROFILE\_TIMER can only be used once in a function. To declare two timers in a C function, use TAU\_PROFILE\_DECLARE\_TIMER and TAU\_PROFILE\_CREATE\_TIMER.

## Example

C :

```
int f1(void) {
    TAU_PROFILE_DECLARE_TIMER(t1);
    TAU_PROFILE_DECLARE_TIMER(t2);

    TAU_PROFILE_CREATE_TIMER(t1, "timer1", "", TAU_USER);
    TAU_PROFILE_CREATE_TIMER(t2, "timer2", "", TAU_USER);

    TAU_PROFILE_START(t1);
    ...
    TAU_PROFILE_START(t2);
    ...
    TAU_PROFILE_STOP(t2);
    TAU_PROFILE_STOP(t1);
    return 0;
}
```

## See Also

TAU\_PROFILE\_CREATE\_TIMER

---

## Name

TAU\_PROFILE\_CREATE\_TIMER -- Creates a timer for C

C:

```
TAU_PROFILE_CREATE_TIMER(timer);
Profiler timer;
```

## Description

Because C89 does not allow mixed code and declarations, TAU\_PROFILE\_TIMER can only be used once in a function. To declare two timers in a C function, use TAU\_PROFILE\_DECLARE\_TIMER and TAU\_PROFILE\_CREATE\_TIMER.

## Example

C :

```
int f1(void) {
    TAU_PROFILE_DECLARE_TIMER(t1);
    TAU_PROFILE_DECLARE_TIMER(t2);

    TAU_PROFILE_CREATE_TIMER(t1, "timer1", "", TAU_USER);
    TAU_PROFILE_CREATE_TIMER(t2, "timer2", "", TAU_USER);

    TAU_PROFILE_START(t1);
    ...
    TAU_PROFILE_START(t2);
    ...
    TAU_PROFILE_STOP(t2);
    TAU_PROFILE_STOP(t1);
    return 0;
}
```

## See Also

TAU\_PROFILE\_DECLARE\_TIMER, TAU\_PROFILE\_START, TAU\_PROFILE\_STOP

---

## Name

TAU\_GLOBAL\_TIMER -- Declares a global timer

C/C++:

```
TAU_GLOBAL_TIMER(timer, function_name, type, group);
Profiler timer;
char* or string& function_name;
char* or string& type;
TauGroup_t group;
```

## Description

As TAU\_PROFILE\_TIMER is used within the scope of a block (typically a routine), TAU\_GLOBAL\_TIMER can be used across different routines.

## Example

C/C++ :

```
/* f1.c */
TAU_GLOBAL_TIMER(globalTimer, "global timer", "", TAU_USER);

/* f2.c */

TAU_GLOBAL_TIMER_EXTERNAL(globalTimer);
int foo(void) {
    TAU_GLOBAL_TIMER_START(globalTimer);
    /* ... */
    TAU_GLOBAL_TIMER_STOP();
}
```

## See Also

TAU\_GLOBAL\_TIMER\_EXTERNAL,  
TAU\_GLOBAL\_TIMER\_STOP

TAU\_GLOBAL\_TIMER\_START,

---

## Name

TAU\_GLOBAL\_TIMER\_EXTERNAL -- Declares a global timer from an external compilation unit

C/C++:

```
TAU_GLOBAL_TIMER_EXTERNAL(timer);
Profiler timer;
```

## Description

TAU\_GLOBAL\_TIMER\_EXTERNAL allows you to access a timer defined in another compilation unit.

## Example

C/C++:

```
/* f1.c */
TAU_GLOBAL_TIMER(globalTimer, "global timer", "", TAU_USER);
/* f2.c */

TAU_GLOBAL_TIMER_EXTERNAL(globalTimer);
int foo(void) {
    TAU_GLOBAL_TIMER_START(globalTimer);
    /* ... */
    TAU_GLOBAL_TIMER_STOP();
}
```

## See Also

TAU\_GLOBAL\_TIMER, TAU\_GLOBAL\_TIMER\_START, TAU\_GLOBAL\_TIMER\_STOP

---

## Name

TAU\_GLOBAL\_TIMER\_START -- Starts a global timer

C/C++:

```
TAU_GLOBAL_TIMER_START(timer);
Profiler timer;
```

## Description

TAU\_GLOBAL\_TIMER\_START starts a global timer.

## Example

C/C++:

```
/* f1.c */
TAU_GLOBAL_TIMER(globalTimer, "global timer", "", TAU_USER);
/* f2.c */

TAU_GLOBAL_TIMER_EXTERNAL(globalTimer);
int foo(void) {
    TAU_GLOBAL_TIMER_START(globalTimer);
    /* ... */
    TAU_GLOBAL_TIMER_STOP();
}
```

## See Also

TAU\_GLOBAL\_TIMER, TAU\_GLOBAL\_TIMER\_EXTERNAL, TAU\_GLOBAL\_TIMER\_STOP

---

## Name

TAU\_GLOBAL\_TIMER\_STOP -- Stops a global timer

C/C++:

```
TAU_GLOBAL_TIMER_STOP( );
```

## Description

TAU\_GLOBAL\_TIMER\_STOP stops a global timer.

## Example

C/C++:

```
/* f1.c */
TAU_GLOBAL_TIMER(globalTimer, "global timer", "", TAU_USER);
/* f2.c */
TAU_GLOBAL_TIMER_EXTERNAL(globalTimer);
int foo(void) {
    TAU_GLOBAL_TIMER_START(globalTimer);
    /* ... */
    TAU_GLOBAL_TIMER_STOP();
}
```

## See Also

TAU\_GLOBAL\_TIMER, TAU\_GLOBAL\_TIMER\_EXTERNAL, TAU\_GLOBAL\_TIMER\_START

---

## Name

TAU\_PHASE -- Profile a C++ function as a phase

```
TAU_PHASE(function_name, type, group);  
char* or string& function_name;  
char* or string& type;  
TauGroup_t group;
```

## Description

TAU\_PHASE profiles a function as a phase. This macro defines the function and takes care of the timer start and stop as well. The timer will stop when the macro goes out of scope (as in C++ destruction).

## Example

```
int foo(char *str) {  
    TAU_PHASE(foo, "int (char *)", TAU_DEFAULT);  
    ...  
}
```

## See Also

TAU\_PHASE\_CREATE\_DYNAMIC, TAU\_PHASE\_CREATE\_STATIC

---

## Name

TAU\_DYNAMIC\_PHASE -- Defines a dynamic phase.

C/C++:

```
TAU_DYNAMIC_PHASE(phase, function_name, type, group);  
Phase phase;  
char* or string& function_name;  
char* or string& type;  
TauGroup_t group;
```

Fortran:

```
TAU_DYNAMIC_PHASE(phase, name);  
integer phase(2);  
character name(size);
```

## Description

TAU\_DYNAMIC\_PHASE creates a dynamic phase. The name of the timer can be different for each execution.

## Example

C/C++:

```
int main(int argc, char **argv) {  
    int i;  
    TAU_PROFILE_TIMER(t, "main()", "", TAU_DEFAULT);  
    TAU_PROFILE_SET_NODE(0);  
    TAU_PROFILE_START(t);  
  
    for (i=0; i<5; i++) {  
        char buf[32];  
        sprintf(buf, "Iteration %d", i);  
  
        TAU_DYNAMIC_PHASE(timer, buf, "", TAU_USER);  
        TAU_PHASE_START(timer);  
        printf("Iteration %d\n", i);  
        f1();  
  
        TAU_PHASE_STOP(timer);  
    }  
    return 0;  
}
```

Fortran :

```
subroutine ITERATION(val)  
    integer val  
    character(13) cvar  
    integer profiler(2) / 0, 0 /  
    save profiler
```

```
print *, "Iteration ", val
write (cvar,'(a9,i2)') 'Iteration', val
call TAU_DYNAMIC_PHASE(profiler, cvar)
call TAU_PHASE_START(profiler)

call F1()
call TAU_PHASE_STOP(profiler)
return
end
```

## See Also

[TAU\\_PHASE\\_CREATE\\_DYNAMIC](#),  
[TAU\\_DYNAMIC\\_PHASE\\_STOP](#)

[TAU\\_DYNAMIC\\_PHASE\\_START](#),

---

## Name

TAU\_PHASE\_CREATE\_DYNAMIC -- Defines a dynamic phase.

C/C++:

```
TAU_PHASE_CREATE_DYNAMIC(phase, function_name, type, group);  
Phase phase;  
char* or string& function_name;  
char* or string& type;  
TauGroup_t group;
```

Fortran:

```
TAU_PHASE_CREATE_DYNAMIC(phase, name);  
integer phase(2);  
character name(size);
```

## Description

TAU\_PHASE\_CREATE\_DYNAMIC creates a dynamic phase. The name of the timer can be different for each execution.

## Example

C/C++:

```
int main(int argc, char **argv) {  
    int i;  
    TAU_PROFILE_TIMER(t, "main()", "", TAU_DEFAULT);  
    TAU_PROFILE_SET_NODE(0);  
    TAU_PROFILE_START(t);  
  
    for (i=0; i<5; i++) {  
        char buf[32];  
        sprintf(buf, "Iteration %d", i);  
  
        TAU_PHASE_CREATE_DYNAMIC(timer, buf, "", TAU_USER);  
        TAU_PHASE_START(timer);  
        printf("Iteration %d\n", i);  
        f1();  
  
        TAU_PHASE_STOP(timer);  
    }  
    return 0;  
}
```

Fortran :

```
subroutine ITERATION(val)  
    integer val  
    character(13) cvar  
    integer profiler(2) / 0, 0 /  
    save profiler
```

```
print *, "Iteration ", val
write (cvar,'(a9,i2)') 'Iteration', val
call TAU_PHASE_CREATE_DYNAMIC(profiler, cvar)
call TAU_PHASE_START(profiler)

call F1()
call TAU_PHASE_STOP(profiler)
return
end
```

## See Also

TAU\_PHASE\_CREATE\_STATIC, TAU\_PHASE\_START, TAU\_PHASE\_STOP

---

## Name

TAU\_PHASE\_CREATE\_STATIC -- Defines a static phase.

C/C++:

```
TAU_PHASE_CREATE_STATIC(phase, function_name, type, group);  
Phase phase;  
char* or string& function_name;  
char* or string& type;  
TauGroup_t group;
```

Fortran:

```
TAU_PHASE_CREATE_STATIC(phase, name);  
integer phase(2);  
character name(size);
```

## Description

TAU\_PHASE\_CREATE\_STATIC creates a static phase. Static phases (and timers) are more efficient than dynamic ones because the function registration only takes place once.

## Example

C/C++ :

```
int f2(void)  
{  
    TAU_PHASE_CREATE_STATIC(t2,"IO Phase", "", TAU_USER);  
    TAU_PHASE_START(t2);  
    input();  
    output();  
    TAU_PHASE_STOP(t2);  
    return 0;  
}
```

Fortran :

```
subroutine F2()  
  
    integer phase(2) / 0, 0 /  
    save    phase  
  
    call TAU_PHASE_CREATE_STATIC(phase,'IO Phase')  
    call TAU_PHASE_START(phase)  
  
    call INPUT()  
    call OUTPUT()  
  
    call TAU_PHASE_STOP(phase)  
end
```

>Python:

```
import pytau
ptr = pytau.phase("foo")

pytau.start(ptr)
foo(2)
pytau.stop(ptr)
```

## See Also

[TAU\\_PHASE\\_CREATE\\_DYNAMIC](#), [TAU\\_PHASE\\_START](#), [TAU\\_PHASE\\_STOP](#)

---

## Name

TAU\_PHASE\_START -- Enters a phase.

C/C++:

```
TAU_PHASE_START(phase);
Phase phase;
```

Fortran:

```
TAU_PHASE_START(phase);
integer phase(2);
```

## Description

TAU\_PHASE\_START enters a phase. Phases can be nested, but not overlapped.

## Example

C/C++ :

```
int f2(void)
{
    TAU_PHASE_CREATE_STATIC(t2,"IO Phase", "", TAU_USER);
    TAU_PHASE_START(t2);
    input();
    output();
    TAU_PHASE_STOP(t2);
    return 0;
}
```

Fortran :

```
subroutine F2()
    integer phase(2) / 0, 0 /
    save    phase

    call TAU_PHASE_CREATE_STATIC(phase,'IO Phase')
    call TAU_PHASE_START(phase)

    call INPUT()
    call OUTPUT()

    call TAU_PHASE_STOP(phase)
end
```

## See Also

TAU\_PHASE\_CREATE\_STATIC, TAU\_PHASE\_CREATE\_DYNAMIC, TAU\_PHASE\_STOP

---

## Name

TAU\_PHASE\_STOP -- Exits a phase.

C/C++:

```
TAU_PHASE_STOP(phase);  
Phase phase;
```

Fortran:

```
TAU_PHASE_STOP(phase);  
integer phase(2);
```

## Description

TAU\_PHASE\_STOP exits a phase. Phases can be nested, but not overlapped.

## Example

C/C++ :

```
int f2(void)  
{  
    TAU_PHASE_CREATE_STATIC(t2,"IO Phase", "", TAU_USER);  
    TAU_PHASE_START(t2);  
    input();  
    output();  
    TAU_PHASE_STOP(t2);  
    return 0;  
}
```

Fortran :

```
subroutine F2()  
  
    integer phase(2) / 0, 0 /  
    save    phase  
  
    call TAU_PHASE_CREATE_STATIC(phase,'IO Phase')  
    call TAU_PHASE_START(phase)  
  
    call INPUT()  
    call OUTPUT()  
  
    call TAU_PHASE_STOP(phase)  
end
```

## See Also

TAU\_PHASE\_CREATE\_STATIC, TAU\_PHASE\_CREATE\_DYNAMIC, TAU\_PHASE\_START

---

## Name

TAU\_DYNAMIC\_PHASE\_START -- Enters a DYNAMIC\_PHASE.

C/C++:

```
TAU_DYNAMIC_PHASE_START(name);
string name;
```

Fortran:

```
TAU_DYNAMIC_PHASE_START(name);
char name(size);
```

## Description

TAU\_DYNAMIC\_PHASE\_START enters a DYNAMIC phase. Phases can be nested, but not overlapped.

## Example

C/C++ :

```
TAU_PROFILE("int foo(int) [{foo.cpp} {13,1}-{20,1}]", " ", TAU_USER);

printf("inside foo: calling bar: x = %d\n", x);
printf("before calling bar in foo\n");
TAU_DYNAMIC_PHASE_START("foo_bar");
bar(x-1); /* 17 */
printf("after calling bar in foo\n");
TAU_DYNAMIC_PHASE_STOP("foo_bar");
return x;
```

Fortran :

```
call TAU_PROFILE_TIMER(profiler, 'FOO [{foo.f90} {8,18}]')
call TAU_PROFILE_START(profiler)
print *, "inside foo: calling bar, x = ", x
  call TAU_DYNAMIC_PHASE_START("foo_bar")
    call bar(x-1)
  print *, "after calling bar"
    call TAU_DYNAMIC_PHASE_STOP("foo_bar");
call TAU_PROFILE_STOP(profiler)
```

## See Also

TAU\_PHASE\_CREATE\_DYNAMIC, TAU\_PHASE\_CREATE\_STATIC, TAU\_PHASE\_STOP

---

## Name

TAU\_DYNAMIC\_PHASE\_STOP -- Enters a DYNAMIC\_PHASE.

C/C++:

```
TAU_DYNAMIC_PHASE_STOP(name);
string name;
```

Fortran:

```
TAU_DYNAMIC_PHASE_STOP(name);
char name(size);
```

## Description

TAU\_DYNAMIC\_PHASE\_STOP leaves a DYNAMIC phase. Phases can be nested, but not overlapped.

## Example

C/C++ :

```
TAU_PROFILE("int foo(int) [{foo.cpp} {13,1}-{20,1}]", " ", TAU_USER);

printf("inside foo: calling bar: x = %d\n", x);
printf("before calling bar in foo\n");
TAU_DYNAMIC_PHASE_START("foo_bar");
bar(x-1); /* 17 */
printf("after calling bar in foo\n");
TAU_DYNAMIC_PHASE_STOP("foo_bar");
return x;
```

Fortran :

```
call TAU_PROFILE_TIMER(profiler, 'FOO [{foo.f90} {8,18}]')
call TAU_PROFILE_START(profiler)
print *, "inside foo: calling bar, x = ", x
  call TAU_DYNAMIC_PHASE_START("foo_bar")
  call bar(x-1)
print *, "after calling bar"
  call TAU_DYNAMIC_PHASE_STOP("foo_bar");
call TAU_PROFILE_STOP(profiler)
```

## See Also

TAU\_PHASE\_CREATE\_STATIC, TAU\_PHASE\_CREATE\_DYNAMIC, TAU\_PHASE\_STOP

---

## Name

TAU\_STATIC\_PHASE\_START -- Enters a STATIC\_PHASE.

C/C++:

```
TAU_STATIC_PHASE_START(name);
string name;
```

Fortran:

```
TAU_STATIC_PHASE_START(name);
char name(size);
```

## Description

TAU\_STATIC\_PHASE\_START enters a static phase. Phases can be nested, but not overlapped.

## Example

C/C++ :

```
TAU_PROFILE("int foo(int) [{foo.cpp} {13,1}-{20,1}]", " ", TAU_USER);

printf("inside foo: calling bar: x = %d\n", x);
printf("before calling bar in foo\n");
TAU_STATIC_PHASE_START("foo_bar");
bar(x-1); /* 17 */
printf("after calling bar in foo\n");
TAU_STATIC_PHASE_STOP("foo_bar");
return x;
```

Fortran :

```
call TAU_PROFILE_TIMER(profiler, 'FOO [{foo.f90} {8,18}]')
call TAU_PROFILE_START(profiler)
print *, "inside foo: calling bar, x = ", x
  call TAU_STATIC_PHASE_START("foo_bar");
  call bar(x-1)
print *, "after calling bar"
  call TAU_STATIC_PHASE_STOP("foo_bar");
call TAU_PROFILE_STOP(profiler)
```

## See Also

TAU\_PHASE\_CREATE\_STATIC, TAU\_PHASE\_CREATE\_DYNAMIC, TAU\_PHASE\_STOP

---

## Name

TAU\_STATIC\_PHASE\_STOP -- Enters a STATIC\_PHASE.

C/C++:

```
TAU_STATIC_PHASE_STOP(name);
string name;
```

Fortran:

```
TAU_STATIC_PHASE_STOP(name);
char name(size);
```

## Description

TAU\_STATIC\_PHASE\_STOP leaves a static phase. Phases can be nested, but not overlapped.

## Example

C/C++ :

```
TAU_PROFILE("int foo(int) [{foo.cpp} {13,1}-{20,1}]", " ", TAU_USER);

printf("inside foo: calling bar: x = %d\n", x);
printf("before calling bar in foo\n");
TAU_STATIC_PHASE_START("foo_bar");
bar(x-1); /* 17 */
printf("after calling bar in foo\n");
TAU_STATIC_PHASE_STOP("foo_bar");
return x;
```

Fortran :

```
call TAU_PROFILE_TIMER(profiler, 'FOO [{foo.f90} {8,18}]')
call TAU_PROFILE_START(profiler)
print *, "inside foo: calling bar, x = ", x
  call TAU_STATIC_PHASE_START("foo_bar");
  call bar(x-1)
print *, "after calling bar"
  call TAU_STATIC_PHASE_STOP("foo_bar");
call TAU_PROFILE_STOP(profiler)
```

## See Also

TAU\_PHASE\_CREATE\_STATIC, TAU\_PHASE\_CREATE\_DYNAMIC, TAU\_PHASE\_STOP

---

## Name

TAU\_GLOBAL\_PHASE -- Declares a global phase

C/C++:

```
TAU_GLOBAL_PHASE(phase, function_name, type, group);  
Phase phase;  
char* or string& function_name;  
char* or string& type;  
TauGroup_t group;
```

## Description

Declares a global phase to be used in multiple compilation units.

## Example

C/C++:

```
/* f1.c */  
  
TAU_GLOBAL_PHASE(globalPhase, "global phase", "", TAU_USER);  
  
/* f2.c */  
  
int bar(void) {  
    TAU_GLOBAL_PHASE_START(globalPhase);  
    /* ... */  
    TAU_GLOBAL_PHASE_STOP(globalPhase);  
}
```

## See Also

TAU\_GLOBAL\_PHASE\_EXTERNAL,  
TAU\_GLOBAL\_PHASE\_STOP

TAU\_GLOBAL\_PHASE\_START,

---

## Name

TAU\_GLOBAL\_PHASE\_EXTERNAL -- Declares a global phase from an external compilation unit

C/C++:

```
TAU_GLOBAL_PHASE_EXTERNAL(timer);
Profiler timer;
```

## Description

TAU\_GLOBAL\_PHASE\_EXTERNAL allows you to access a phase defined in another compilation unit.

## Example

C/C++:

```
/* f1.c */
TAU_GLOBAL_PHASE(globalPhase, "global phase", "", TAU_USER);
/* f2.c */

int bar(void) {
    TAU_GLOBAL_PHASE_START(globalPhase);
    /* ... */
    TAU_GLOBAL_PHASE_STOP(globalPhase);
}
```

## See Also

[TAU\\_GLOBAL\\_PHASE](#), [TAU\\_GLOBAL\\_PHASE\\_START](#), [TAU\\_GLOBAL\\_PHASE\\_STOP](#)

---

## Name

TAU\_GLOBAL\_PHASE\_START -- Starts a global phase

C/C++:

```
TAU_GLOBAL_PHASE_START(phase);
Phase phase;
```

## Description

TAU\_GLOBAL\_PHASE\_START starts a global phase.

## Example

C/C++:

```
/* f1.c */
TAU_GLOBAL_PHASE(globalPhase, "global phase", "", TAU_USER);
/* f2.c */

int bar(void) {
    TAU_GLOBAL_PHASE_START(globalPhase);
    /* ... */
    TAU_GLOBAL_PHASE_STOP(globalPhase);
}
```

## See Also

TAU\_GLOBAL\_PHASE, TAU\_GLOBAL\_PHASE\_EXTERNAL, TAU\_GLOBAL\_PHASE\_STOP

---

## Name

TAU\_GLOBAL\_PHASE\_STOP -- Stops a global phase

C/C++:

```
TAU_GLOBAL_PHASE_STOP(phase);  
Phase phase;
```

## Description

TAU\_GLOBAL\_PHASE\_STOP stops a global phase.

## Example

C/C++:

```
/* f1.c */  
  
TAU_GLOBAL_PHASE(globalPhase, "global phase", "", TAU_USER);  
  
/* f2.c */  
  
int bar(void) {  
    TAU_GLOBAL_PHASE_STOP(globalPhase);  
    /* ... */  
    TAU_GLOBAL_PHASE_STOP(globalPhase);  
}
```

## See Also

TAU\_GLOBAL\_PHASE, TAU\_GLOBAL\_PHASE\_EXTERNAL, TAU\_GLOBAL\_PHASE\_START

---

## Name

TAU\_PROFILE\_EXIT -- Alerts the profiling system to an exit call

C/C++:

```
TAU_PROFILE_EXIT(message);
const char * message;
```

Fortran:

```
TAU_PROFILE_EXIT(message);
character message(size);
```

## Description

TAU\_PROFILE\_EXIT should be called prior to an error exit from the program so that any profiles or event traces can be dumped to disk before quitting.

## Example

C/C++ :

```
if ((ret = open(...)) < 0) {
    TAU_PROFILE_EXIT("ERROR in opening a file");
    perror("open() failed");
    exit(1);
}
```

Fortran :

```
call TAU_PROFILE_EXIT('abort called')
```

## See Also

TAU\_DB\_DUMP

---

## Name

TAU\_REGISTER\_THREAD -- Register a thread with the profiling system

C/C++:

```
TAU_REGISTER_THREAD();
```

Fortran:

```
TAU_REGISTER_THREAD();
```

## Description

To register a thread with the profiling system, invoke the TAU\_REGISTER\_THREAD macro in the run method of the thread prior to executing any other TAU macro. This sets up thread identifiers that are later used by the instrumentation system.

## Example

C/C++:

```
void * threaded_func(void *data) {
    TAU_REGISTER_THREAD();
    { /* **** NOTE WE START ANOTHER BLOCK IN THREAD */
        TAU_PROFILE_TIMER(tautimer, "threaded_func()", "int ()",
                          TAU_DEFAULT);
        TAU_PROFILE_START(tautimer);
        work(); /* work done by this thread */
        TAU_PROFILE_STOP(tautimer);
    }
    return NULL;
}
```

Fortran :

```
call TAU_REGISTER_THREAD()
```

## Caveat

PDT based tau\_instrumentor does not insert TAU\_REGISTER\_THREAD calls, they must be inserted manually

---

## Name

TAU\_PROFILE\_GET\_NODE -- Returns the measurement system's node id

C/C++:

```
TAU_PROFILE_GET_NODE(node);
int node;
```

Fortran:

```
TAU_PROFILE_GET_NODE(node);
integer node;
```

## Description

TAU\_PROFILE\_GET\_NODE gives the node id for the processes in which it is called. When using MPI node id is the same as MPI rank.

## Example

C/C++ :

```
int main (int argc, char **argv) {
    int nodeid;
    TAU_PROFILE_GET_NODE(nodeid);
    return 0;
}
```

Fortran :

```
PROGRAM SUM_OF_CUBES
  INTEGER :: N
  call TAU_PROFILE_GET_NODE(N)
END PROGRAM SUM_OF_CUBES
```

Python:

```
import pytau
pytau.setNode(0)
```

## See Also

TAU\_PROFILE\_GET\_CONTEXT

---

## Name

TAU\_PROFILE\_GET\_CONTEXT -- Gives the measurement system's context id

C/C++:

```
TAU_PROFILE_GET_CONTEXT(context);
int context;
```

Fortran:

```
TAU_PROFILE_GET_CONTEXT(context);
integer context;
```

## Description

TAU\_PROFILE\_GET\_CONTEXT gives the context id for the processes in which it is called.

## Example

C/C++ :

```
int main (int argc, char **argv) {
    int i;
    TAU_PROFILE_GET_CONTEXT(i);
    return 0;
}
```

Fortran :

```
PROGRAM SUM_OF_CUBES
  INTEGER :: C
  call TAU_PROFILE_GET_CONTEXT(C)
END PROGRAM SUM_OF_CUBES
```

## See Also

TAU\_PROFILE\_SET\_CONTEXT

---

## Name

TAU\_PROFILE\_SET\_THREAD -- Informs the measurement system of the THREAD id

C/C++:

```
TAU_PROFILE_SET_THREAD(THREAD);  
int THREAD;
```

Fortran:

```
TAU_PROFILE_SET_THREAD(THREAD);  
integer THREAD;
```

## Description

The TAU\_PROFILE\_SET\_THREAD macro sets the thread identifier of the executing task for profiling and tracing. Tasks are identified using node, context and thread ids. The profile data files generated will accordingly be named profile.<THREAD>.<context>.<thread>. Note that it is not necessary to call TAU\_PROFILE\_SET\_THREAD when you configured with a threading package (including OpenMP).

## Example

C/C++ :

```
int main (int argc, char **argv) {  
    int ret, i;  
    pthread_attr_t attr;  
    pthread_t tid;  
    TAU_PROFILE_TIMER(tautimer,"main()", "int (int, char **)",  
                      TAU_DEFAULT);  
    TAU_PROFILE_START(tautimer);  
    TAU_PROFILE_INIT(argc, argv);  
    TAU_PROFILE_SET_THREAD(0);  
    /* ... */  
    TAU_PROFILE_STOP(tautimer);  
    return 0;  
}
```

Fortran :

```
PROGRAM SUM_OF_CUBES  
    integer profiler(2) / 0, 0 /  
    save profiler  
    INTEGER :: H, T, U  
    call TAU_PROFILE_INIT()  
    call TAU_PROFILE_TIMER(profiler, 'PROGRAM SUM_OF_CUBES')  
    call TAU_PROFILE_START(profiler)  
    call TAU_PROFILE_SET_THREAD(0)  
    ! This program prints all 3-digit numbers that  
    ! equal the sum of the cubes of their digits.  
    DO H = 1, 9  
        DO T = 0, 9  
            DO U = 0, 9
```

```
IF (100*H + 10*T + U == H**3 + T**3 + U**3) THEN
    PRINT "(3I1)", H, T, U
ENDIF
END DO
END DO
call TAU_PROFILE_STOP(profiler)
END PROGRAM SUM_OF_CUBES
```

**Python:**

```
import pytau
pytau.setThread(0)
```

## See Also

TAU\_PROFILE\_SET\_NODE TAU\_PROFILE\_SET\_CONTEXT

---

## Name

TAU\_PROFILE\_GET\_THREAD -- Gives the measurement system's thread id

C/C++:

```
TAU_PROFILE_GET_THREAD(thread);
int thread;
```

Fortran:

```
TAU_PROFILE_GET_THREAD(THREAD);
integer THREAD;
```

## Description

TAU\_PROFILE\_GET\_THREAD gives the thread id for the processes in which it is called.

## Example

C/C++ :

```
int main (int argc, char **argv) {
    int i;
    TAU_PROFILE_GET_THREAD(i);
    return 0;
}
```

Fortran :

```
PROGRAM SUM_OF_CUBES
  INTEGER :: T
  call TAU_PROFILE_GET_THREAD(T)
! This program prints all 3-digit numbers that
! equal the sum of the cubes of their digits.
END PROGRAM SUM_OF_CUBES
```

Python:

```
import pytau
pytau.getThread(i)
```

## See Also

TAU\_PROFILE\_GET\_NODE TAU\_PROFILE\_GET\_CONTEXT

---

## Name

TAU\_PROFILE\_SET\_NODE -- Informs the measurement system of the node id

C/C++:

```
TAU_PROFILE_SET_NODE(node);
int node;
```

Fortran:

```
TAU_PROFILE_SET_NODE(node);
integer node;
```

## Description

The TAU\_PROFILE\_SET\_NODE macro sets the node identifier of the executing task for profiling and tracing. Tasks are identified using node, context and thread ids. The profile data files generated will accordingly be named profile.<node>.<context>.<thread>. Note that it is not necessary to call TAU\_PROFILE\_SET\_NODE when using the TAU MPI wrapper library.

## Example

C/C++ :

```
int main (int argc, char **argv) {
    int ret, i;
    pthread_attr_t attr;
    pthread_t tid;
    TAU_PROFILE_TIMER(tautimer,"main()", "int (int, char **)",
                      TAU_DEFAULT);
    TAU_PROFILE_START(tautimer);
    TAU_PROFILE_INIT(argc, argv);
    TAU_PROFILE_SET_NODE(0);
    /* ... */
    TAU_PROFILE_STOP(tautimer);
    return 0;
}
```

Fortran :

```
PROGRAM SUM_OF_CUBES
    integer profiler(2) / 0, 0 /
    save profiler
    INTEGER :: H, T, U
    call TAU_PROFILE_INIT()
    call TAU_PROFILE_TIMER(profiler, 'PROGRAM SUM_OF_CUBES')
    call TAU_PROFILE_START(profiler)
    call TAU_PROFILE_SET_NODE(0)
    ! This program prints all 3-digit numbers that
    ! equal the sum of the cubes of their digits.
    DO H = 1, 9
        DO T = 0, 9
            DO U = 0, 9
```

```
IF (100*H + 10*T + U == H**3 + T**3 + U**3) THEN
    PRINT "(3I1)", H, T, U
ENDIF
END DO
END DO
call TAU_PROFILE_STOP(profiler)
END PROGRAM SUM_OF_CUBES
```

**Python:**

```
import pytau
pytau.setNode(0)
```

## See Also

[TAU\\_PROFILE\\_SET\\_CONTEXT](#)

---

## Name

TAU\_PROFILE\_SET\_CONTEXT -- Informs the measurement system of the context id

C/C++:

```
TAU_PROFILE_SET_CONTEXT(context);
int context;
```

Fortran:

```
TAU_PROFILE_SET_CONTEXT(context);
integer context;
```

## Description

The TAU\_PROFILE\_SET\_CONTEXT macro sets the context identifier of the executing task for profiling and tracing. Tasks are identified using context, context and thread ids. The profile data files generated will accordingly be named profile.<context>.<context>.<thread>. Note that it is not necessary to call TAU\_PROFILE\_SET\_CONTEXT when using the TAU MPI wrapper library.

## Example

C/C++ :

```
int main (int argc, char **argv) {
    int ret, i;
    pthread_attr_t attr;
    pthread_t tid;
    TAU_PROFILE_TIMER(tautimer,"main()", "int (int, char **)",
                      TAU_DEFAULT);
    TAU_PROFILE_START(tautimer);
    TAU_PROFILE_INIT(argc, argv);
    TAU_PROFILE_SET_NODE(0);
    TAU_PROFILE_SET_CONTEXT(1);
    /* ... */
    TAU_PROFILE_STOP(tautimer);
    return 0;
}
```

Fortran :

```
PROGRAM SUM_OF_CUBES
    integer profiler(2) / 0, 0 /
    save profiler
    INTEGER :: H, T, U
    call TAU_PROFILE_INIT()
    call TAU_PROFILE_TIMER(profiler, 'PROGRAM SUM_OF_CUBES')
    call TAU_PROFILE_START(profiler)
    call TAU_PROFILE_SET_NODE(0)
    call TAU_PROFILE_SET_CONTEXT(1)
    ! This program prints all 3-digit numbers that
    ! equal the sum of the cubes of their digits.
    DO H = 1, 9
```

```
DO T = 0, 9
  DO U = 0, 9
    IF (100*H + 10*T + U == H**3 + T**3 + U**3) THEN
      PRINT "(3I1)", H, T, U
    ENDIF
    END DO
  END DO
END DO
call TAU_PROFILE_STOP(profiler)
END PROGRAM SUM_OF_CUBES
```

## See Also

[TAU\\_PROFILE\\_SET\\_NODE](#)

---

## Name

TAU\_REGISTER\_FORK -- Informs the measurement system that a fork has taken place

C/C++:

```
TAU_REGISTER_FORK(pid, option);
int pid;
enum TauFork_t option;
```

## Description

To register a child process obtained from the fork() syscall, invoke the TAU\_REGISTER\_FORK macro. It takes two parameters, the first is the node id of the child process (typically the process id returned by the fork call or any 0..N-1 range integer). The second parameter specifies whether the performance data for the child process should be derived from the parent at the time of fork (TAU\_INCLUDE\_PARENT\_DATA) or should be independent of its parent at the time of fork (TAU\_EXCLUDE\_PARENT\_DATA). If the process id is used as the node id, before any analysis is done, all profile files should be converted to contiguous node numbers (from 0..N-1). It is highly recommended to use flat contiguous node numbers in this call for profiling and tracing.

## Example

C/C++ :

```
pID = fork();
if (pID == 0) {
    printf("Parent : pid returned %d\n", pID)
} else {
    // If we'd used the TAU_INCLUDE_PARENT_DATA, we get
    // the performance data from the parent in this process
    // as well.
    TAU_REGISTER_FORK(pID, TAU_EXCLUDE_PARENT_DATA);
    printf("Child : pid = %d", pID);
}
```

---

## Name

TAU\_REGISTER\_EVENT -- Registers a user event

C/C++:

```
TAU_REGISTER_EVENT(variable, event_name);
TauUserEvent variable;
char *event_name;
```

Fortran:

```
TAU_REGISTER_EVENT(variable, event_name);
int variable(2);
character event_name(size);
```

## Description

TAU can profile user-defined events using TAU\_REGISTER\_EVENT. The meaning of the event is determined by the user. The first argument to TAU\_REGISTER\_EVENT is the pointer to an integer array. This array is declared with a save attribute as shown below.

## Example

C/C++ :

```
int user_square(int count) {
    TAU_REGISTER_EVENT(uel, "UserSquare Event");
    TAU_EVENT(uel, count * count);
    return 0;
}
```

Fortran :

```
integer eventid(2)
save eventid
call TAU_REGISTER_EVENT(eventid, 'Error in Iteration')
call TAU_EVENT(eventid, count)
```

## See Also

TAU\_EVENT, TAU\_REGISTER\_CONTEXT\_EVENT, TAU\_REPORT\_STATISTICS,  
TAU\_REPORT\_THREAD\_STATISTICS, TAU\_GET\_EVENT\_NAMES, TAU\_GET\_EVENT\_VALS

---

## Name

TAU\_PROFILER\_REGISTER\_EVENT -- Registers a user event

C/C++:

```
TAU_PROFILER_REGISTER_EVENT(variable, event, event_name);  
TauUserEvent variable;  
void *event;  
char *event_name;
```

Fortran:

```
TAU_PROFILER_REGISTER_EVENT(integer , event_name);  
integer eventid(2);  
character event_name(size);
```

## Description

TAU can profile user-defined events using TAU\_PROFILER\_REGISTER\_EVENT. The meaning of the event is determined by the user. The first argument to TAU\_PROFILER\_REGISTER\_EVENT is the pointer to an integer array. This array is declared with a save attribute as shown below.

## Example

C/C++ :

```
int user_square(int count) {  
    void *uel;  
    TAU_PROFILER_REGISTER_EVENT(uel, "UserSquare Event");  
    TAU_EVENT(uel, count * count);  
    return 0;  
}
```

Fortran :

```
integer eventid(2)  
save eventid  
call TAU_PROFILER_REGISTER_EVENT(eventid, 'Error in Iteration')  
call TAU_EVENT(eventid, count)
```

## See Also

TAU\_EVENT, TAU\_REGISTER\_CONTEXT\_EVENT, TAU\_REPORT\_STATISTICS,  
TAU\_REPORT\_THREAD\_STATISTICS, TAU\_GET\_EVENT\_NAMES, TAU\_GET\_EVENT\_VALS

---

## Name

TAU\_EVENT -- Triggers a user event

C/C++:

```
TAU_EVENT(variable, value);
TauUserEvent variable;
double value;
```

Fortran:

```
TAU_EVENT(variable, value);
integer variable(2);
real value;
```

## Description

Triggers an event that was registered with TAU\_REGISTER\_EVENT.

## Example

C/C++ :

```
int user_square(int count) {
    TAU_REGISTER_EVENT(uel, "UserSquare Event");
    TAU_EVENT(uel, count * count);
    return 0;
}
```

Fortran :

```
integer eventid(2)
save eventid
call TAU_REGISTER_EVENT(eventid, 'Error in Iteration')
call TAU_EVENT(eventid, count)
```

## See Also

TAU\_REGISTER\_EVENT

---

## Name

TAU\_EVENT\_THREAD -- Triggers a user event on a given thread

C/C++:

```
TAU_EVENT_THREAD(variable, value, thread id);
TauUserEVENT_THREAD variable;
double value;
int thread id;
```

Fortran:

```
TAU_EVENT_THREAD(variable, value, thread id);
integer variable(2);
real value;
integer thread id;
```

## Description

Triggers an event that was registered with TAU\_REGISTER\_EVENT on a given thread.

## Example

C/C++:

```
int user_square(int count) {
    TAU_REGISTER_EVENT(uel, "UserSquare Event");
    TAU_EVENT_THREAD(uel, count * count, threadid);
    return 0;
}
```

Fortran :

```
integer eventid(2)
save eventid
call TAU_REGISTER_EVENT(eventid, 'Error in Iteration')
call TAU_EVENT_THREAD(eventid, count, threadid)
```

## See Also

TAU\_REGISTER\_EVENT

---

## Name

TAU\_REGISTER\_CONTEXT\_EVENT -- Registers a context event

C/C++:

```
TAU_REGISTER_CONTEXT_EVENT(variable, event_name);
TauUserEvent variable;
char *event_name;
```

Fortran:

```
TAU_REGISTER_CONTEXT_EVENT(variable, event_name);
int variable(2);
character event_name(size);
```

## Description

Creates a context event with name. A context event appends the names of routines executing on the callstack to the name specified by the user. Whenever a context event is triggered, the callstack is examined to determine the context of execution. Starting from the parent function where the event is triggered, TAU walks up the callstack to a depth specified by the user in the environment variable TAU\_CALLPATH\_DEPTH . If this environment variable is not specified, TAU uses 2 as the default depth. For e.g., if the user registers a context event with the name "memory used" and specifies 3 as the callpath depth, and if the event is triggered in two locations (in routine a, when it was called by b, when it was called by c, and in routine h, when it was called by g, when it was called by i), then, we'd see the user defined event information for "memory used: c() => b() => a()" and "memory used: i() => g() => h()".

## Example

C/C++ :

```
int f2(void)
{
    static int count = 0;
    count++;
    TAU_PROFILE("f2()", "(sleeps 2 sec, calls f3)", TAU_USER);
    TAU_REGISTER_CONTEXT_EVENT(event, "Iteration count");
/*
    if (count == 2)
        TAU_DISABLE_CONTEXT_EVENT(event);
*/
    printf("Inside f2: sleeps 2 sec, calls f3\n");

    TAU_CONTEXT_EVENT(event, 232+count);
    sleep(2);
    f3();
    return 0;
}
```

Fortran :

```
subroutine foo(id)
  integer id

  integer profiler(2) / 0, 0 /
  integer maev(2) / 0, 0 /
  integer mdev(2) / 0, 0 /
  save profiler, maev, mdev

  integer :: ierr
  integer :: h, t, u
  INTEGER, ALLOCATABLE :: STORAGEARY(:)
  DOUBLEPRECISION edata

  call TAU_PROFILE_TIMER(profiler, 'FOO')
  call TAU_PROFILE_START(profiler)
  call TAU_PROFILE_SET_NODE(0)

  call TAU_REGISTER_CONTEXT_EVENT(maev, "STORAGEARY Alloc [cubes.f:20]")
  call TAU_REGISTER_CONTEXT_EVENT(mdev, "STORAGEARY Dealloc [cubes.f:37]")

  allocate(STORAGEARY(1:999), STAT=IERR)
  edata = SIZE(STORAGEARY)*sizeof(INTEGER)
  call TAU_CONTEXT_EVENT(maev, edata)
  ...
  deallocate(STORAGEARY)
  edata = SIZE(STORAGEARY)*sizeof(INTEGER)
  call TAU_CONTEXT_EVENT(mdev, edata)
  call TAU_PROFILE_STOP(profiler)
end subroutine foo
```

## See Also

TAU\_CONTEXT\_EVENT, TAU\_ENABLE\_CONTEXT\_EVENT,  
TAU\_DISABLE\_CONTEXT\_EVENT, TAU\_REGISTER\_EVENT, TAU\_REPORT\_STATISTICS,  
TAU\_REPORT\_THREAD\_STATISTICS, TAU\_GET\_EVENT\_NAMES, TAU\_GET\_EVENT\_VALS

---

## Name

TAU\_CONTEXT\_EVENT -- Triggers a context event

C/C++:

```
TAU_CONTEXT_EVENT(variable, value);
TauUserEvent variable;
double value;
```

Fortran:

```
TAU_CONTEXT_EVENT(variable, value);
integer variable(2);
real value;
```

## Description

Triggers a context event. A context event associates the name with the list of routines along the call-stack. A context event tracks information like a user defined event and TAU records the maxima, minima, mean, std. deviation and the number of samples for each context event. A context event helps distinguish the data supplied by the user based on the location where an event occurs and the sequence of actions (routine/timer invocations) that preceeded the event. The depth of the the callstack embedded in the context event's name is specified by the user in the environment variable TAU\_CALLPATH\_DEPTH. If this variable is not specified, TAU uses a default depth of 2.

## Example

C/C++:

```
int f2(void)
{
    static int count = 0;
    count++;
    TAU_PROFILE("f2()", "(sleeps 2 sec, calls f3)", TAU_USER);
    TAU_REGISTER_CONTEXT_EVENT(event, "Iteration count");
/*
    if (count == 2)
        TAU_DISABLE_CONTEXT_EVENT(event);
*/
    printf("Inside f2: sleeps 2 sec, calls f3\n");

    TAU_CONTEXT_EVENT(event, 232+count);
    sleep(2);
    f3();
    return 0;
}
```

Fortran :

```
integer memevent(2) / 0, 0 /
save memevent
call TAU_REGISTER_CONTEXT_EVENT(memevent, "STORAGEARY mem allocated")
```

```
call TAU_CONTEXT_EVENT(memevent, SIZEOF(STORAGEARY)*sizeof( INTEGER))
```

## See Also

[TAU\\_REGISTER\\_CONTEXT\\_EVENT](#)

---

## Name

TAU\_ENABLE\_CONTEXT\_EVENT -- Enable a context event

C/C++:

```
TAU_ENABLE_CONTEXT_EVENT(event);
TauUserEvent event;
```

## Description

Enables a context event.

## Example

C/C++:

```
int f2(void) {
    static int count = 0;
    count++;
    TAU_PROFILE("f2()", "(sleeps 2 sec, calls f3)", TAU_USER);
    TAU_REGISTER_CONTEXT_EVENT(event, "Iteration count");

    if (count == 2)
        TAU_DISABLE_CONTEXT_EVENT(event);
    else
        TAU_ENABLE_CONTEXT_EVENT(event);

    printf("Inside f2: sleeps 2 sec, calls f3\n");
    TAU_CONTEXT_EVENT(event, 232+count);
    sleep(2);
    f3();
    return 0;
}
```

## See Also

TAU\_REGISTER\_CONTEXT\_EVENT, TAU\_DISABLE\_CONTEXT\_EVENT

---

## Name

TAU\_DISABLE\_CONTEXT\_EVENT -- Disable a context event

C/C++:

```
TAU_DISABLE_CONTEXT_EVENT(event);
TauUserEvent event;
```

## Description

Disables a context event.

## Example

C/C++:

```
int f2(void) {
    static int count = 0;
    count++;
    TAU_PROFILE("f2()", "(sleeps 2 sec, calls f3)", TAU_USER);
    TAU_REGISTER_CONTEXT_EVENT(event, "Iteration count");

    if (count == 2)
        TAU_DISABLE_CONTEXT_EVENT(event);
    else
        TAU_ENABLE_CONTEXT_EVENT(event);

    printf("Inside f2: sleeps 2 sec, calls f3\n");

    TAU_CONTEXT_EVENT(event, 232+count);
    sleep(2);
    f3();
    return 0;
}
```

## See Also

TAU\_REGISTER\_CONTEXT\_EVENT, TAU\_ENABLE\_CONTEXT\_EVENT

---

## Name

TAU\_EVENT\_SET\_NAME -- Sets the name of an event

C/C++:

```
TAU_EVENT_SET_NAME(event, name);
TauUserEvent event;
const char *name;
```

## Description

Changes the name of an event.

## Example

C/C++ :

```
TAU_EVENT_SET_NAME(event, "new name");
```

## See Also

TAU\_REGISTER\_EVENT

---

## Name

TAU\_EVENT\_DISABLE\_MAX -- Disables tracking of maximum statistic for a given event

C/C++:

```
TAU_EVENT_DISABLE_MAX(event);  
TauUserEvent event;
```

## Description

Disables tracking of maximum statistic for a given event

## Example

C/C++ :

```
TAU_EVENT_DISABLE_MAX(event);
```

## See Also

[TAU\\_REGISTER\\_EVENT](#)

---

## Name

TAU\_EVENT\_DISABLE\_MEAN -- Disables tracking of mean statistic for a given event

C/C++:

```
TAU_EVENT_DISABLE_MEAN(event);
TauUserEvent event;
```

## Description

Disables tracking of mean statistic for a given event

## Example

C/C++ :

```
TAU_EVENT_DISABLE_MEAN(event);
```

## See Also

[TAU\\_REGISTER\\_EVENT](#)

---

## Name

TAU\_EVENT\_DISABLE\_MIN -- Disables tracking of minimum statistic for a given event

C/C++:

```
TAU_EVENT_DISABLE_MIN(event);  
TauUserEvent event;
```

## Description

Disables tracking of minimum statistic for a given event

## Example

C/C++ :

```
TAU_EVENT_DISABLE_MIN(event);
```

## See Also

[TAU\\_REGISTER\\_EVENT](#)

---

## Name

TAU\_EVENT\_DISABLE\_STDDEV -- Disables tracking of standard deviation statistic for a given event

C/C++:

```
TAU_EVENT_DISABLE_STDDEV(event);
TauUserEvent event;
```

## Description

Disables tracking of standard deviation statistic for a given event

## Example

C/C++ :

```
TAU_EVENT_DISABLE_STDDEV(event);
```

## See Also

TAU\_REGISTER\_EVENT

---

## Name

TAU\_REPORT\_STATISTICS -- Outputs statistics

C/C++:

```
TAU_REPORT_STATISTICS();
```

Fortran:

```
TAU_REPORT_STATISTICS();
```

## Description

TAU\_REPORT\_STATISTICS prints the aggregate statistics of user events across all threads in each node. Typically, this should be called just before the main thread exits.

## Example

C/C++ :

```
TAU_REPORT_STATISTICS();
```

Fortran :

```
call TAU_REPORT_STATISTICS()
```

## See Also

TAU\_REGISTER\_EVENT,  
TAU\_REPORT\_THREAD\_STATISTICS

TAU\_REGISTER\_CONTEXT\_EVENT,

---

## Name

`TAU_REPORT_THREAD_STATISTICS` -- Outputs statistics, plus thread statistics

C/C++:

```
TAU_REPORT_THREAD_STATISTICS();
```

Fortran:

```
TAU_REPORT_THREAD_STATISTICS();
```

## Description

`TAU_REPORT_THREAD_STATISTICS` prints the aggregate, as well as per thread user event statistics. Typically, this should be called just before the main thread exits.

## Example

C/C++ :

```
TAU_REPORT_THREAD_STATISTICS();
```

Fortran :

```
call TAU_REPORT_THREAD_STATISTICS()
```

## See Also

`TAU_REGISTER_EVENT`, `TAU_REGISTER_CONTEXT_EVENT`, `TAU_REPORT_STATISTICS`

---

## Name

TAU\_ENABLE\_INSTRUMENTATION -- Enables instrumentation

C/C++:

```
TAU_ENABLE_INSTRUMENTATION();
```

Fortran:

```
TAU_ENABLE_INSTRUMENTATION();
```

## Description

TAU\_ENABLE\_INSTRUMENTATION macro re-enables all TAU instrumentation. All instances of functions and statements that occur between the disable/enable section are ignored by TAU. This allows a user to limit the trace size, if the macros are used to disable recording of a set of iterations that have the same characteristics as, for example, the first recorded instance.

## Example

C/C++:

```
int main(int argc, char **argv) {
    foo();
    TAU_DISABLE_INSTRUMENTATION();
    for (int i = 0; i < N; i++) {
        bar(); // not recorded
    }
    TAU_ENABLE_INSTRUMENTATION();
    bar(); // recorded
}
```

Fortran :

```
call TAU_DISABLE_INSTRUMENTATION()
...
call TAU_ENABLE_INSTRUMENTATION()
```

Python:

```
import pytau
pytau.enableInstrumentation()
...
pytau.disableInstrumentation()
```

## See Also

---

TAU\_DISABLE\_INSTRUMENTATION, TAU\_ENABLE\_GROUP, TAU\_DISABLE\_GROUP,  
TAU\_INIT, TAU\_PROFILE\_INIT

---

## Name

TAU\_DISABLE\_INSTRUMENTATION -- Disables instrumentation

C/C++:

```
TAU_DISABLE_INSTRUMENTATION();
```

Fortran:

```
TAU_DISABLE_INSTRUMENTATION();
```

## Description

TAU\_DISABLE\_INSTRUMENTATION macro disables all entry/exit instrumentation within all threads of a context. This allows the user to selectively enable and disable instrumentation in parts of his/her code. It is important to re-enable the instrumentation within the same basic block and scope.

## Example

C/C++:

```
int main(int argc, char **argv) {
    foo();
    TAU_DISABLE_INSTRUMENTATION();
    for (int i = 0; i < N; i++) {
        bar(); // not recorded
    }
    TAU_DISABLE_INSTRUMENTATION();
    bar(); // recorded
}
```

Fortran :

```
call TAU_DISABLE_INSTRUMENTATION()
...
call TAU_DISABLE_INSTRUMENTATION()
```

Python:

```
import pytau
pytau.enableInstrumentation()
...
pytau.disableInstrumentation()
```

## See Also

---

TAU\_ENABLE\_INSTRUMENTATION,    TAU\_ENABLE\_GROUP,    TAU\_DISABLE\_GROUP,  
TAU\_INIT, TAU\_PROFILE\_INIT

---

## Name

TAU\_ENABLE\_GROUP -- Enables tracking of a given group

C/C++:

```
TAU_ENABLE_GROUP(group);
TauGroup_t group;
```

Fortran:

```
TAU_ENABLE_GROUP(group);
integer group;
```

## Description

Enables the instrumentation for a given group. By default, it is already on.

## Example

C/C++ :

```
void foo() {
    TAU_PROFILE("foo()", " ", TAU_USER);
    ...
    TAU_ENABLE_GROUP(TAU_USER);
}
```

Fortran :

```
include 'Profile/TauFAPI.h'
call TAU_ENABLE_GROUP(TAU_USER)
```

Python:

```
import pytau
pytau.enableGroup(TAU_USER)
```

## See Also

TAU\_ENABLE\_INSTRUMENTATION, TAU\_DISABLE\_INSTRUMENTATION,  
TAU\_DISABLE\_GROUP, TAU\_INIT, TAU\_PROFILE\_INIT

---

## Name

TAU\_DISABLE\_GROUP -- Disables tracking of a given group

C/C++:

```
TAU_DISABLE_GROUP(group);
TauGroup_t group;
```

Fortran:

```
TAU_DISABLE_GROUP(group);
integer group;
```

## Description

Disables the instrumentation for a given group. By default, it is on.

## Example

C/C++ :

```
void foo() {
    TAU_PROFILE("foo()", " ", TAU_USER);
    ...
    TAU_DISABLE_GROUP(TAU_USER);
}
```

Fortran :

```
include 'Profile/TauFAPI.h'
call TAU_DISABLE_GROUP(TAU_USER)
```

Python:

```
import pytau
pytau.disableGroup(TAU_USER)
```

## See Also

TAU\_ENABLE\_INSTRUMENTATION, TAU\_DISABLE\_INSTRUMENTATION,  
TAU\_ENABLE\_GROUP, TAU\_INIT, TAU\_PROFILE\_INIT

---

## Name

TAU\_PROFILE\_TIMER\_SET\_GROUP -- Change the group of a timer

C/C++:

```
TAU_PROFILE_TIMER_SET_GROUP(timer, group);
Profiler timer;
TauGroup_t group;
```

## Description

TAU\_PROFILE\_TIMER\_SET\_GROUP changes the group associated with a timer.

## Example

C/C++ :

```
void foo() {
    TAU_PROFILE_TIMER(t, "foo loop timer", " ", TAU_USER1);
    ...
    TAU_PROFILE_TIMER_SET_GROUP(t, TAU_USER3);
}
```

## See Also

TAU\_PROFILE\_TIMER, TAU\_PROFILE\_TIMER\_SET\_GROUP\_NAME

---

## Name

TAU\_PROFILE\_TIMER\_SET\_GROUP\_NAME -- Changes the group name for a timer

C/C++:

```
TAU_PROFILE_TIMER_SET_GROUP_NAME(timer, groupname);
Profiler timer;
char *groupname;
```

## Description

TAU\_PROFILE\_TIMER\_SET\_GROUP\_NAME changes the group name associated with a given timer.

## Example

C/C++ :

```
void foo() {
    TAU_PROFILE_TIMER(looptimer, "foo: loop1", " ", TAU_USER);
    TAU_PROFILE_START(looptimer);
    for (int i = 0; i < N; i++) { /* do something */ }
    TAU_PROFILE_STOP(looptimer);
    TAU_PROFILE_TIMER_SET_GROUP_NAME("Field");
}
```

## See Also

TAU\_PROFILE\_TIMER, TAU\_PROFILE\_TIMER\_SET\_GROUP

---

## Name

TAU\_PROFILE\_TIMER\_SET\_NAME -- Changes the name of a timer

C/C++:

```
TAU_PROFILE_TIMER_SET_NAME(timer, newname);
Profiler timer;
string newname;
```

## Description

TAU\_PROFILE\_TIMER\_SET\_NAME macro changes the name associated with a timer to the newname argument.

## Example

C/C++:

```
void foo() {
    TAU_PROFILE_TIMER(timer1, "foo:loop1", " ", TAU_USER);
    ...
    TAU_PROFILE_TIMER_SET_NAME(timer1, "foo:lines 21-34");
}
```

## See Also

TAU\_PROFILE\_TIMER

---

## Name

TAU\_PROFILE\_TIMER\_SET\_TYPE -- Changes the type of a timer

C/C++:

```
TAU_PROFILE_TIMER_SET_TYPE(timer, newname);
Profiler timer;
string newname;
```

## Description

TAU\_PROFILE\_TIMER\_SET\_TYPE macro changes the type associated with a timer to the newname argument.

## Example

C/C++:

```
void foo() {
    TAU_PROFILE_TIMER(timer1, "foo", "int", TAU_USER);
    ...
    TAU_PROFILE_TIMER_SET_TYPE(timer1, "long");
}
```

## See Also

TAU\_PROFILE\_TIMER

---

## Name

TAU\_PROFILE\_SET\_GROUP\_NAME -- Changes the group name of a profiled section

C/C++:

```
TAU_PROFILE_SET_GROUP_NAME(groupname);  
char *groupname;
```

## Description

TAU\_PROFILE\_SET\_GROUP\_NAME macro allows the user to change the group name associated with the instrumented routine. This macro must be called within the instrumented routine.

## Example

C/C++ :

```
void foo() {  
    TAU_PROFILE("foo()", "void ()", TAU_USER);  
    TAU_PROFILE_SET_GROUP_NAME("Particle");  
    /* gives a more meaningful group name */  
}
```

## See Also

TAU\_PROFILE

---

## Name

TAU\_INIT -- Processes command-line arguments for selective instrumentation

C/C++:

```
TAU_INIT(argc, argv);
int *argc;
char ***argv;
```

## Description

TAU\_INIT parses and removes the command-line arguments for the names of profile groups that are to be selectively enabled for instrumentation. By default, if this macro is not used, functions belonging to all profile groups are enabled. TAU\_INIT differs from TAU\_PROFILE\_INIT only in the argument types.

## Example

C/C++:

```
int main(int argc, char **argv) {
    TAU_PROFILE("main()", "int (int, char **)", TAU_GROUP_12);
    TAU_INIT(&argc, &argv);
    ...
}
% ./a.out --profile 12+14
```

## See Also

[TAU\\_PROFILE\\_INIT](#)

---

## Name

TAU\_PROFILE\_INIT -- Processes command-line arguments for selective instrumentation

C/C++:

```
TAU_PROFILE_INIT(argc, argv);
int argc;
char **argv;
```

Fortran:

```
TAU_PROFILE_INIT();
```

## Description

TAU\_PROFILE\_INIT parses the command-line arguments for the names of profile groups that are to be selectively enabled for instrumentation. By default, if this macro is not used, functions belonging to all profile groups are enabled. TAU\_INIT differs from TAU\_PROFILE\_INIT only in the argument types.

## Example

C/C++ :

```
int main(int argc, char **argv) {
    TAU_PROFILE("main()", "int (int, char **)", TAU_DEFAULT);
    TAU_PROFILE_INIT(argc, argv);
    ...
}
% ./a.out --profile 12+14
```

Fortran :

```
PROGRAM SUM_OF_CUBES
    integer profiler(2)
    save profiler

    call TAU_PROFILE_INIT()
    ...
```

## See Also

TAU\_INIT

---

## Name

TAU\_GET\_PROFILE\_GROUP -- Creates groups based on names

C/C++:

```
TAU_GET_PROFILE_GROUP(groupname);  
char *groupname;
```

## Description

TAU\_GET\_PROFILE\_GROUP allows the user to dynamically create groups based on strings, rather than use predefined, statically assigned groups such as TAU\_USER1, TAU\_USER2 etc. This allows names to be associated in creating unique groups that are more meaningful, using names of files or directories for instance.

## Example

C/C++ :

```
#define PARTICLES TAU_GET_PROFILE_GROUP( "PARTICLES" )  
  
void foo() {  
    TAU_PROFILE("foo()", " ", PARTICLES);  
}  
  
void bar() {  
    TAU_PROFILE("bar()", " ", PARTICLES);  
}
```

Python:

```
import pytau  
  
pytau.getProfileGroup( "PARTICLES" )
```

## See Also

TAU\_ENABLE\_GROUP\_NAME, TAU\_DISABLE\_GROUP\_NAME,  
TAU\_ENABLE\_ALL\_GROUPS, TAU\_DISABLE\_ALL\_GROUPS

---

## Name

TAU\_ENABLE\_GROUP\_NAME -- Enables a group based on name

C/C++:

```
TAU_ENABLE_GROUP_NAME(groupname);  
char *groupname;
```

Fortran:

```
TAU_ENABLE_GROUP_NAME(groupname);  
character groupname(size);
```

## Description

TAU\_ENABLE\_GROUP\_NAME macro can turn on the instrumentation associated with routines based on a dynamic group assigned to them. It is important to note that this and the TAU\_DISABLE\_GROUP\_NAME macros apply to groups created dynamically using TAU\_GET\_PROFILE\_GROUP.

## Example

C/C++ :

```
/* tau_instrumentor was invoked with -g DTM for a set of files */  
TAU_DISABLE_GROUP_NAME("DTM");  
dtm_routines();  
/* disable and then re-enable the group with the name DTM */  
TAU_ENABLE_GROUP_NAME("DTM");
```

Fortran :

```
! tau_instrumentor was invoked with -g DTM for this file  
call TAU_PROFILE_TIMER(profiler, "ITERATE>DTM")  
  
call TAU_DISABLE_GROUP_NAME("DTM")  
! Disable, then re-enable DTM group  
call TAU_ENABLE_GROUP_NAME("DTM")
```

Python:

```
import pytau  
pytau.enableGroupName("DTM")
```

## See Also

---

TAU\_GET\_PROFILE\_GROUP, TAU\_DISABLE\_GROUP\_NAME, TAU\_ENABLE\_ALL\_GROUPS,  
TAU\_DISABLE\_ALL\_GROUPS

---

## Name

TAU\_DISABLE\_GROUP\_NAME -- Disables a group based on name

C/C++:

```
TAU_DISABLE_GROUP_NAME(groupname);
char *groupname;
```

Fortran:

```
TAU_DISABLE_GROUP_NAME(groupname);
character groupname(size);
```

## Description

Similar to TAU\_ENABLE\_GROUP\_NAME , this macro turns off the instrumentation in all routines associated with the dynamic group created using the tau\_instrumentor -g <group\_name> argument.

## Example

C/C++ :

```
/* tau_instrumentor was invoked with -g DTM for a set of files */
TAU_DISABLE_GROUP_NAME("DTM");
dtm_routines();
/* disable and then re-enable the group with the name DTM */
TAU_ENABLE_GROUP_NAME("DTM");
```

Fortran :

```
! tau_instrumentor was invoked with -g DTM for this file
call TAU_PROFILE_TIMER(profiler, "ITERATE>DTM")
call TAU_DISABLE_GROUP_NAME("DTM")
! Disable, then re-enable DTM group
call TAU_ENABLE_GROUP_NAME("DTM")
```

Python:

```
import pytau
pytau.disableGroupName("DTM")
```

## See Also

TAU\_GET\_PROFILE\_GROUP, TAU\_ENABLE\_GROUP\_NAME, TAU\_ENABLE\_ALL\_GROUPS,  
TAU\_DISABLE\_ALL\_GROUPS

---

## Name

TAU\_ENABLE\_ALL\_GROUPS -- Enables instrumentation in all groups

C/C++:

```
TAU_ENABLE_ALL_GROUPS();
```

Fortran:

```
TAU_ENABLE_ALL_GROUPS();
```

## Description

This macro turns on instrumentation in all groups

## Example

C/C++ :

```
TAU_ENABLE_ALL_GROUPS();
```

Fortran :

```
call TAU_ENABLE_ALL_GROUPS();
```

Python:

```
import pytau
pytau.enableAllGroups()
```

## See Also

TAU\_GET\_PROFILE\_GROUP, TAU\_ENABLE\_GROUP\_NAME,  
TAU\_DISABLE\_GROUP\_NAME, TAU\_DISABLE\_ALL\_GROUPS

---

## Name

TAU\_DISABLE\_ALL\_GROUPS -- Disables instrumentation in all groups

C/C++:

```
TAU_DISABLE_ALL_GROUPS( );
```

Fortran:

```
TAU_DISABLE_ALL_GROUPS( );
```

## Description

This macro turns off instrumentation in all groups.

## Example

C/C++ :

```
void foo() {
    TAU_DISABLE_ALL_GROUPS();
    TAU_ENABLE_GROUP_NAME( "PARTICLES" );
}
```

Fortran :

```
call TAU_DISABLE_ALL_GROUPS( );
```

Python:

```
import pytau
pytau.disableAllGroups()
```

## See Also

TAU\_GET\_PROFILE\_GROUP, TAU\_ENABLE\_GROUP\_NAME,  
TAU\_DISABLE\_GROUP\_NAME, TAU\_ENABLE\_ALL\_GROUPS

---

## Name

TAU\_GET\_EVENT\_NAMES -- Gets the registered user events.

C/C++:

```
TAU_GET_EVENT_NAMES(eventList, numEvents);
const char ***eventList;
int *numEvents;
```

## Description

Retrieves user event names for all user-defined events

## Example

C/C++ :

```
const char **eventList;
int numEvents;

TAU_GET_EVENT_NAMES(eventList, numEvents);

cout << "numEvents: " << numEvents << endl;
```

## See Also

TAU\_REGISTER\_EVENT, TAU\_REGISTER\_CONTEXT\_EVENT, TAU\_GET\_EVENT\_VALS

---

## Name

TAU\_GET\_EVENT\_VALS -- Gets user event data for given user events.

C/C++:

```
TAU_GET_EVENT_VALS(inUserEvents, numUserEvents, numEvents, max, min,
mean, sumSqe);
const char **inUserEvents;
int numUserEvents;
int **numEvents;
double **max;
double **min;
double **mean;
double **sumSqe;
```

## Description

Retrieves user defined event data for the specified user defined events. The list of events are specified by the first parameter (eventList) and the user specifies the number of events in the second parameter (numUserEvents). TAU returns the number of times the event was invoked in the numUserEvents. The max, min, mean values are returned in the following parameters. TAU computes the sum of squares of the given event and returns this value in the next argument (sumSqe).

## Example

C/C++:

```
const char **eventList;
int numEvents;

TAU_GET_EVENT_NAMES(eventList, numEvents);

cout << "numEvents: " << numEvents << endl;

if (numEvents > 0) {
    int *numSamples;
    double *max;
    double *min;
    double *mean;
    double *sumSqr;

    TAU_GET_EVENT_VALS(eventList, numEvents, numSamples,
    max, min, mean, sumSqr);
    for (int i=0; i<numEvents; i++) {
        cout << "-----\n";
        cout << "User Event: " << eventList[i] << endl;
        cout << "Number of Samples: " << numSamples[i] << endl;
        cout << "Maximum Value: " << max[i] << endl;
        cout << "Minimum Value: " << min[i] << endl;
        cout << "Mean Value: " << mean[i] << endl;
        cout << "Sum Squared: " << sumSqr[i] << endl;
    }
}
```

## See Also

TAU\_REGISTER\_EVENT, TAU\_REGISTER\_CONTEXT\_EVENT, TAU\_GET\_EVENT\_NAMES

---

## Name

TAU\_GET\_COUNTER\_NAMES -- Gets the counter names

C/C++:

```
TAU_GET_COUNTER_NAMES(counterList, numCounters);
char **counterList;
int numCounters;
```

## Description

TAU\_GET\_COUNTER\_NAMES returns the list of counter names and the number of counters used for measurement. When wallclock time is used, the counter name of "default" is returned.

## Example

C/C++:

```
int numOfCounters;
const char ** counterList;

TAU_GET_COUNTER_NAMES(counterList, numOfCounters);

for(int j=0;j<numOfCounters;j++){
    cout << "The counter names so far are: " << counterList[j] << endl;
}
```

Python:

```
import pytau

pytau.getCounterNames(counterList, numOfCounters);
```

## See Also

TAU\_GET\_FUNC NAMES, TAU\_GET\_FUNC\_VALS

---

## Name

TAU\_GET\_FUNC\_NAMES -- Gets the function names

C/C++:

```
TAU_GET_FUNC_NAMES(functionList, numFuncs);
char **functionList;
int numFuncs;
```

## Description

This macro fills the funcList argument with the list of timer and routine names. It also records the number of routines active in the numFuncs argument.

## Example

C/C++:

```
const char ** functionList;
int numFunctions;

TAU_GET_FUNC_NAMES(functionList, numFunctions);

for(int i=0;i<numFunctions;i++){
    cout << "This function names so far are: " << functionList[i] << endl;
}
```

*Python:*

```
import pytau

pytau.getFuncNames(functionList, numFunctions)
```

## See Also

TAU\_GET\_COUNTER\_NAMES, TAU\_GET\_FUNC\_VALS, TAU\_DUMP\_FUNC\_NAMES,  
TAU\_DUMP\_FUNC\_VALS

---

## Name

TAU\_GET\_FUNC\_VALS -- Gets detailed performance data for given functions

C/C++:

```
TAU_GET_FUNC_VALS(inFuncs, numOfFuncs, counterExclusiveValues, counterInclusiveValues, numOfCalls, numOfSubRoutines, counterNames, numOfCounters, tid);
const char **inFuncs;
int numOfFuncs;
double ***counterExclusiveValues;
double ***counterInclusiveValues;
int **numOfCalls;
int **numOfSubRoutines;
const char ***counterNames;
int *numOfCounters;
int tid;
```

## Description

It gets detailed performance data for the list of routines. The user specifies inFuncs and the number of routines; TAU then returns the other arguments with the performance data. counterExclusiveValues and counterInclusiveValues are two dimensional arrays: the first dimension is the routine id and the second is counter id. The value is indexed by these two dimensions. numCalls and numSubrs (or child routines) are one dimensional arrays.

## Example

C/C++:

```
const char **inFuncs;
/* The first dimension is functions, and the
second dimension is counters */
double **counterExclusiveValues;
double **counterInclusiveValues;
int *numOfCalls;
int *numOfSubRoutines;
const char **counterNames;
int numOfCouns;

TAU_GET_FUNC_NAMES(functionList, numOfFunctions);

/* We are only interested in the first two routines
that are executing in this context. So, we allocate
space for two routine names and get the performance
data for these two routines at runtime. */
if (numOfFunctions >= 2) {
    inFuncs = (const char **) malloc(sizeof(const char *) * 2);

    inFuncs[0] = functionList[0];
    inFuncs[1] = functionList[1];

    //Just to show consistency.
    TAU_DB_DUMP();

    TAU_GET_FUNC_VALS(inFuncs, 2,
        counterExclusiveValues,
```

```
    counterInclusiveValues,
    numOfCalls,
    numOfSubRoutines,
    counterNames,
    numOfCouns);

    TAU_DUMP_FUNC_VALS_INCR(inFuncs, 2);

    cout << "@@@@@@@@@@@@@" << endl;
    cout << "The number of counters is: " << numOfCouns << endl;
    cout << "The first counter is: " << counterNames[0] << endl;

    cout << "The Exclusive value of: " << inFuncs[0]
    << " is: " << counterExclusiveValues[0][0] << endl;
    cout << "The numOfSubRoutines of: " << inFuncs[0]
    << " is: " << numOfSubRoutines[0]
    << endl;

    cout << "The Inclusive value of: " << inFuncs[1]
    << " is: " << counterInclusiveValues[1][0]
    << endl;
    cout << "The numOfCalls of: " << inFuncs[1]
    << " is: " << numOfCalls[1]
    << endl;

    cout << "@@@@@@@@@@@@@" << endl;
}

TAU_DB_DUMP_INCR();
```

Python:

```
import pytau

pytau.dumpFuncVals("foo", "bar", "bar2")
```

## See Also

[TAU\\_GET\\_COUNTER\\_NAMES](#), [TAU\\_GET\\_FUNC\\_NAMES](#), [TAU\\_DUMP\\_FUNC\\_NAMES](#),  
[TAU\\_DUMP\\_FUNC\\_VALS](#)

---

## Name

TAU\_ENABLE\_TRACKING\_MEMORY -- Enables memory tracking

C/C++:

```
TAU_ENABLE_TRACKING_MEMORY( );
```

Fortran:

```
TAU_ENABLE_TRACKING_MEMORY( );
```

## Description

Enables tracking of the heap memory utilization in the program. TAU takes a sample of the heap memory utilized (as reported by the mallinfo system call) and associates it with a single global user defined event. An interrupt is generated every 10 seconds and the value of the heap memory used is recorded in the user defined event. The inter-interrupt interval (default of 10 seconds) may be set by the user using the call TAU\_SET\_INTERRUPT\_INTERVAL.

## Example

C/C++ :

```
TAU_ENABLE_TRACKING_MEMORY( );
```

Fortran :

```
call TAU_ENABLE_TRACKING_MEMORY( )
```

Python:

```
import pytau
pytau.enableTrackingMemory()
```

## See Also

TAU\_DISABLE\_TRACKING\_MEMORY, TAU\_SET\_INTERRUPT\_INTERVAL,  
TAU\_TRACK\_MEMORY, TAU\_TRACK\_MEMORY\_HERE

---

## Name

TAU\_DISABLE\_TRACKING\_MEMORY -- Disables memory tracking

C/C++:

```
TAU_DISABLE_TRACKING_MEMORY( );
```

Fortran:

```
TAU_DISABLE_TRACKING_MEMORY( );
```

## Description

Disables tracking of heap memory utilization. This call may be used in sections of code where TAU should not interrupt the execution to periodically track the heap memory utilization.

## Example

C/C++ :

```
TAU_DISABLE_TRACKING_MEMORY( );
```

Fortran :

```
call TAU_DISABLE_TRACKING_MEMORY( )
```

Python:

```
import pytau
pytau.disableTrackingMemory()
```

## See Also

TAU\_ENABLE\_TRACKING\_MEMORY, TAU\_SET\_INTERRUPT\_INTERVAL,  
TAU\_TRACK\_MEMORY, TAU\_TRACK\_MEMORY\_HERE

---

## Name

TAU\_TRACK\_MEMORY -- Initializes memory tracking system

C/C++:

```
TAU_TRACK_MEMORY();
```

Fortran:

```
TAU_TRACK_MEMORY();
```

## Description

For memory profiling, there are two modes of operation: 1) the user explicitly inserts TAU\_TRACK\_MEMORY\_HERE() calls in the source code and the memory event is triggered at those locations, and 2) the user enables tracking memory by calling TAU\_TRACK\_MEMORY() and an interrupt is generated every 10 seconds and the memory event is triggered with the current value. Also, this interrupt interval can be changed by calling TAU\_SET\_INTERRUPT\_INTERVAL(value). The tracking of memory events in both cases can be explicitly enabled or disabled by calling the macros TAU\_ENABLE\_TRACKING\_MEMORY() or TAU\_DISABLE\_TRACKING\_MEMORY() respectively.

## Example

C/C++ :

```
TAU_TRACK_MEMORY();
```

Fortran :

```
call TAU_TRACK_MEMORY()
```

Python:

```
import pytau  
pytau.trackMemory()
```

## See Also

TAU\_ENABLE\_TRACKING\_MEMORY,  
TAU\_SET\_INTERRUPT\_INTERVAL,  
TAU\_TRACK\_MEMORY\_HEADROOM

TAU\_DISABLE\_TRACKING\_MEMORY,  
TAU\_TRACK\_MEMORY\_HERE,

---

## Name

TAU\_TRACK\_MEMORY\_HERE -- Triggers memory tracking at a given execution point

C/C++:

```
TAU_TRACK_MEMORY_HERE();
```

Fortran:

```
TAU_TRACK_MEMORY_HERE();
```

## Description

Triggers memory tracking at a given execution point

## Example

C/C++ :

```
int main(int argc, char **argv) {
    TAU_PROFILE("main()", " ", TAU_DEFAULT);
    TAU_PROFILE_SET_NODE(0);

    TAU_TRACK_MEMORY_HERE();

    int *x = new int[5*1024*1024];
    TAU_TRACK_MEMORY_HERE();
    return 0;
}
```

Fortran :

```
INTEGER, ALLOCATABLE :: STORAGEARY(:)
allocate(STORAGEARY(1:999), STAT=IERR)

! if we wish to record a sample of the heap memory
! utilization at this point, invoke the following call:
call TAU_TRACK_MEMORY_HERE()
```

Python:

```
import pytau
pytau.trackMemoryHere()
```

## See Also

---

TAU\_TRACK\_MEMORY

---

## Name

TAU\_ENABLE\_TRACKING\_MEMORY\_HEADROOM -- Enables memory headroom tracking

C/C++:

```
TAU_ENABLE_TRACKING_MEMORY_HEADROOM( );
```

Fortran:

```
TAU_ENABLE_TRACKING_MEMORY_HEADROOM( );
```

## Description

TAU\_ENABLE\_TRACKING\_MEMORY\_HEADROOM( ) enables memory headroom tracking after a TAU\_DISABLE\_TRACKING\_MEMORY\_HEADROOM( ).

## Example

C/C++ :

```
TAU_DISABLE_TRACKING_MEMORY_HEADROOM( );
/* do some work */
...
/* re-enable tracking memory headroom */
TAU_ENABLE_TRACKING_MEMORY_HEADROOM( );
```

**Fortran :**

```
call TAU_ENABLE_TRACKING_MEMORY_HEADROOM( );
```

**Fortran :**

```
import pytau
pytau.enableTrackingMemoryHeadroom()
```

## See Also

TAU\_TRACK\_MEMORY\_HEADROOM, TAU\_DISABLE\_TRACKING\_MEMORY\_HEADROOM,  
TAU\_TRACK\_MEMORY\_HEADROOM\_HERE, TAU\_SET\_INTERRUPT\_INTERVAL

---

## Name

TAU\_DISABLE\_TRACKING\_MEMORY\_HEADROOM -- Disables memory headroom tracking

C/C++:

```
TAU_DISABLE_TRACKING_MEMORY_HEADROOM( );
```

Fortran:

```
TAU_DISABLE_TRACKING_MEMORY_HEADROOM( );
```

## Description

TAU\_DISABLE\_TRACKING\_MEMORY\_HEADROOM( ) disables memory headroom tracking.

## Example

C/C++ :

```
TAU_DISABLE_TRACKING_MEMORY_HEADROOM( );
```

Fortran :

```
call TAU_DISABLE_TRACKING_MEMORY_HEADROOM( )
```

Python:

```
import pytau
pytau.disableTrackingMemoryHeadroom()
```

## See Also

TAU\_TRACK\_MEMORY\_HEADROOM, TAU\_ENABLE\_TRACKING\_MEMORY\_HEADROOM,  
TAU\_TRACK\_MEMORY\_HEADROOM\_HERE, TAU\_SET\_INTERRUPT\_INTERVAL

---

## Name

TAU\_TRACK\_MEMORY\_HEADROOM -- Track the headroom (amount of memory for a process to grow) by periodically interrupting the program

C/C++:

```
TAU_TRACK_MEMORY_HEADROOM( );
```

Fortran:

```
TAU_TRACK_MEMORY_HEADROOM( );
```

## Description

Tracks the amount of memory available for the process before it runs out of free memory on the heap. This call sets up a signal handler that is invoked every 10 seconds by an interrupt (this interval may be altered by using the TAU\_SET\_INTERRUPT\_INTERVAL call). Inside the interrupt handler, TAU evaluates how much memory it can allocate and associates it with the callstack using the TAU context events (See TAU\_REGISTER\_CONTEXT\_EVENT). The user can vary the size of the callstack by setting the environment variable TAU\_CALLPATH\_DEPTH (default is 2). This call is useful on machines like IBM BG/L where no virtual memory (or paging using the swap space) is present. The amount of heap memory available to the program is limited by the amount of available physical memory. TAU executes a series of malloc calls with a granularity of 1MB and determines the amount of memory available for the program to grow.

## Example

C/C++ :

```
TAU_TRACK_MEMORY_HEADROOM( );
```

Fortran :

```
call TAU_TRACK_MEMORY_HEADROOM( )
```

Python:

```
import pytau
pytau.trackMemoryHeadroom()
```

## See Also

TAU\_TRACK\_MEMORY,  
TAU\_ENABLE\_TRACKING\_MEMORY\_HEADROOM,  
TAU\_DISABLE\_TRACKING\_MEMORY\_HEADROOM,

TAU\_SET\_INTERRUPT\_INTERVAL,

TAU\_TRACK\_MEMORY\_HEADROOM\_HERE

---

## Name

TAU\_TRACK\_MEMORY\_HEADROOM\_HERE -- Takes a sample of the amount of memory available at a given point.

C/C++:

```
TAU_TRACK_MEMORY_HEADROOM_HERE();
```

Fortran:

```
TAU_TRACK_MEMORY_HEADROOM_HERE();
```

## Description

Instead of relying on a periodic interrupt to track the amount of memory available to grow, this call may be used to take a sample at a given location in the source code. Context events are used to track the amount of memory headroom.

## Example

C/C++ :

```
ary = new double [1024*1024*50];
TAU_TRACK_MEMORY_HEADROOM_HERE();
```

Fortran :

```
INTEGER, ALLOCATABLE :: STORAGEARY(:)
allocate(STORAGEARY(1:999), STAT=IERR)
TAU_TRACK_MEMORY_HEADROOM_HERE();
```

Python:

```
import pytau
pytau.trackMemoryHeadroomHere()
```

## See Also

TAU\_TRACK\_MEMORY\_HEADROOM

---

## Name

TAU\_SET\_INTERRUPT\_INTERVAL -- Change the inter-interrupt interval for tracking memory and headroom

C/C++:

```
TAU_SET_INTERRUPT_INTERVAL(value);
int value;
```

Fortran:

```
TAU_SET_INTERRUPT_INTERVAL(value);
integer value;
```

## Description

Set the interrupt interval for tracking memory and headroom (See TAU\_TRACK\_MEMORY and TAU\_TRACK\_MEMORY\_HEADROOM). By default an inter-interrupt interval of 10 seconds is used in TAU. This call allows the user to set it to a different value specified by the argument value.

## Example

C/C++ :

```
TAU_SET_INTERRUPT_INTERVAL(2)
/* invokes the interrupt handler for memory every 2s */
```

Fortran :

```
call TAU_SET_INTERRUPT_INTERVAL(2)
```

Python:

```
import pytau
pytau.setInterruptTninterval(2)
```

## See Also

TAU\_TRACK\_MEMORY, TAU\_TRACK\_MEMORY\_HEADROOM

---

## Name

CT -- Returns the type information for a variable

C/C++:

```
CT(variable);
<type> variable;
```

## Description

The CT macro returns the runtime type information string of a variable. This is useful in constructing the type parameter of the TAU\_PROFILE macro. For templates, the type information can be constructed using the type of the return and the type of each of the arguments (parameters) of the template. The example in the following macro will clarify this.

## Example

C/C++ :

```
TAU_PROFILE( "foo::memberfunc( )" , CT(*this) , TAU_DEFAULT );
```

## See Also

TAU\_PROFILE, TAU\_PROFILE\_TIMER, TAU\_TYPE\_STRING

---

## Name

TAU\_TYPE\_STRING -- Creates a type string

C++:

```
TAU_TYPE_STRING(variable, type_string);
string &variable;
string &type_string;
```

## Description

This macro assigns the string constructed in type\_string to the variable. The + operator and the CT macro can be used to construct the type string of an object. This is useful in identifying templates uniquely, as shown below.

## Example

C++:

```
template<class PLayout>
ostream& operator<<(ostream& out, const ParticleBase<PLayout>& P) {
    TAU_TYPE_STRING(taustr, "ostream (ostream, " + CT(P) + " )");
    TAU_PROFILE("operator<<()" taustr, TAU_PARTICLE | TAU_IO);
    ...
}
```

When PLayout is instantiated with "UniformCartesian<3U, double>", this generates the unique template name:

```
operator<<() ostream const
ParticleBase<UniformCartesian<3U, double> > )
```

The following example illustrates the usage of the CT macro to extract the name of the class associated with the given object using CT(\*this);

```
template<class PLayout>
unsigned ParticleBase<PLayout7>::GetMessage(Message& msg, int node) {
    TAU_TYPE_STRING(taustr, CT(*this) + "unsigned (Message, int)");
    TAU_PROFILE("ParticleBase::GetMessage()", taustr, TAU_PARTICLE);
    ...
}
```

When PLayout is instantiated with "UniformCartesian<3U, double>", this generates the unique template name:

```
ParticleBase::GetMessage() ParticleBase<UniformCartesian<3U,
double> > unsigned (Message, int)
```

## See Also

CT, TAU\_PROFILE, TAU\_PROFILE\_TIMER

---

## Name

TAU\_DB\_DUMP -- Dumps the profile database to disk

C/C++:

```
TAU_DB_DUMP();
```

Fortran:

```
TAU_DB_DUMP();
```

## Description

Dumps the profile database to disk. The format of the files is the same as regular profiles, they are simply prefixed with "dump" instead of "profile".

## Example

C/C++ :

```
TAU_DB_DUMP();
```

Fortran :

```
call TAU_DB_DUMP()
```

## See Also

TAU\_DB\_DUMP\_PREFIX,  
TAU\_DUMP\_FUNC\_VALS,  
TAU\_PROFILE\_EXIT

TAU\_DB\_DUMP\_INCR,            TAU\_DUMP\_FUNC\_NAMES,  
TAU\_DUMP\_FUNC\_VALS\_INCR,    TAU\_DB\_PURGE,

---

## Name

TAU\_DB\_MERGED\_DUMP -- Dumps the profile database to disk

C/C++:

```
TAU_DB_MERGED_DUMP( );
```

Fortran:

```
TAU_DB_MERGED_DUMP( );
```

## Description

Dumps the profile database to disk. The format of the files is the same as merged profiles: tauprofile.xml

## Example

C/C++ :

```
TAU_DB_MERGED_DUMP( );
```

Fortran :

```
call TAU_DB_MERGED_DUMP()
```

## See Also

TAU\_DB\_DUMP\_PREFIX,  
TAU\_DUMP\_FUNC\_VALS,  
TAU\_PROFILE\_EXIT

TAU\_DB\_DUMP\_INCR,        TAU\_DUMP\_FUNC\_NAMES,  
TAU\_DUMP\_FUNC\_VALS\_INCR,    TAU\_DB\_PURGE,

---

## Name

TAU\_DB\_DUMP\_INCR -- Dumps profile database into timestamped profiles on disk

C/C++:

```
TAU_DB_DUMP_INCR();
```

## Description

This is similar to the TAU\_DB\_DUMP macro but it produces dump files that have a timestamp in their names. This allows the user to record timestamped incremental dumps as the application executes.

## Example

C/C++ :

```
TAU_DB_DUMP_INCR();
```

Python:

```
import pytau
pytau.dbDumpIncr("prefix")
```

## See Also

TAU\_DB\_DUMP,            TAU\_DB\_DUMP\_PREFIX,            TAU\_DUMP\_FUNC\_NAMES,  
TAU\_DUMP\_FUNC\_VALS,     TAU\_DUMP\_FUNC\_VALS\_INCR,     TAU\_DB\_PURGE,  
TAU\_PROFILE\_EXIT

---

## Name

TAU\_DB\_DUMP\_PREFIX -- Dumps the profile database into profile files with a given prefix

C/C++:

```
TAU_DB_DUMP_PREFIX(prefix);
char *prefix;
```

Fortran:

```
TAU_DB_DUMP_PREFIX(prefix);
character prefix(size);
```

## Description

The TAU\_DB\_DUMP\_PREFIX macro dumps all profile data to disk and records a checkpoint or a snapshot of the profile statistics at that instant. The dump files are named <prefix>.<node>.<context>.<thread>. If prefix is "profile", the files are named profile.0.0.0, etc. and may be read by paraprof/pprof tools as the application executes.

## Example

C/C++ :

```
TAU_DB_DUMP_PREFIX( "prefix" );
```

Fortran :

```
call TAU_DB_DUMP_PREFIX( "prefix" )
```

Python :

```
import pytau
pytau.dbDump( "prefix" )
```

## See Also

TAU\_DB\_DUMP

---

## Name

TAU\_DB\_DUMP\_PREFIX\_TASK -- Dumps the profile database into profile files with a given task

C/C++:

```
TAU_DB_DUMP_PREFIX_TASK(PREFIX_TASK);
char *PREFIX_TASK;
```

Fortran:

```
TAU_DB_DUMP_PREFIX_TASK(prefix, task);
character prefix(size);
integer task(size);
```

## Description

The TAU\_DB\_DUMP\_PREFIX\_TASK macro dumps all profile data to disk and records a checkpoint or a snapshot of the profile statistics on a particular task at that instant. The dump files are named <prefix>.<node>.<context>.<thread>. If prefix is "profile", the files are named profile.0.0.0, etc. and may be read by paraprof/pprof tools as the application executes.

## Example

C/C++ :

```
TAU_DB_DUMP_PREFIX_TASK("PREFIX", taskid);
```

Fortran :

```
call TAU_DB_DUMP_PREFIX_TASK("PREFIX", taskid)
```

Python :

```
import pytau
pytau.dbDump("PREFIX", taskid)
```

## See Also

TAU\_DB\_DUMP\_PREFIX

---

## **Name**

TAU\_DB\_PURGE -- Purges the performance data.

C/C++:

```
TAU_DB_PURGE( );
```

## **Description**

Purges the performance data collected so far.

## **Example**

C/C++ :

```
TAU_DB_PURGE( );
```

## **See Also**

TAU\_DB\_DUMP

---

## Name

TAU\_DUMP\_FUNC\_NAMES -- Dumps function names to disk

C/C++:

```
TAU_DUMP_FUNC_NAMES( );
```

## Description

This macro writes the names of active functions to a file named dump\_functionnames\_<node>.<context>.

## Example

C/C++ :

```
TAU_DUMP_FUNC_NAMES( );
```

Python:

```
import pytau
pytau.dumpFuncNames()
```

## See Also

TAU\_DB\_DUMP, TAU\_DUMP\_FUNC\_VALS, TAU\_DUMP\_FUNC\_VALS\_INCR

---

## Name

TAU\_DUMP\_FUNC\_VALS -- Dumps performance data for given functions to disk.

C/C++:

```
TAU_DUMP_FUNC_VALS(inFuncs, numFuncs);
char **inFuncs;
int numFuncs;
```

## Description

TAU\_DUMP\_FUNC\_VALS writes the data associated with the routines listed in inFuncs to disk. The number of routines is specified by the user in numFuncs.

## Example

C/C++ :

## See Also

TAU\_DB\_DUMP, TAU\_DUMP\_FUNC\_NAMES, TAU\_DUMP\_FUNC\_VALS\_INCR

---

## Name

TAU\_DUMP\_FUNC\_VALS\_INCR -- Dumps function values with a timestamp

C/C++:

```
TAU_DUMP_FUNC_VALS_INCR(inFuncs, numFuncs);
char **inFuncs;
int numFuncs;
```

## Description

Similar to TAU\_DUMP\_FUNC\_VALS. This macro creates an incremental selective dump and dumps the results with a date stamp to the filename such as sel\_dump\_\_Thu-Mar-28-16:30:48-2002\_\_.0.0.0. In this manner the previous TAU\_DUMP\_FUNC\_VALS\_INCR( . . . ) are not overwritten (unless they occur within a second).

## Example

C/C++:

```
const char **inFuncs;
/* The first dimension is functions, and the second dimension is counters */
double **counterExclusiveValues;
double **counterInclusiveValues;
int *numOfCalls;
int *numOfSubRoutines;
const char **counterNames;
int numOfCouns;

TAU_GET_FUNC_VALS(inFuncs, 2,
    counterExclusiveValues,
    counterInclusiveValues,
    numOfCalls,
    numOfSubRoutines,
    counterNames,
    numOfCouns);

TAU_DUMP_FUNC_VALS(inFuncs, 2);
```

Python:

```
import pytau

pytau.dumpFuncValsIncr("foo", "bar", "bar2")
```

## See Also

TAU\_DB\_DUMP, TAU\_DUMP\_FUNC\_NAMES, TAU\_DUMP\_FUNC\_VALS

---

## Name

TAU\_PROFILE\_STMT -- Executes a statement only when TAU is used.

C/C++:

```
TAU_PROFILE_STMT(statement);
statement statement;
```

## Description

TAU\_PROFILE\_STMT executes a statement, or declares a variable that is used only during profiling or for execution of a statement that takes place only when the instrumentation is active. When instrumentation is inactive (i.e., when profiling and tracing are turned off as described in Chapter 2), all macros are defined as null.

## Example

C/C++ :

```
TAU_PROFILE_STMT(T obj); // T is a template parameter)
TAU_TYPE_STRING(str, "void () " + CT(obj) );
```

---

## Name

TAU\_PROFILE\_CALLSTACK -- Generates a callstack trace at a given location.

C/C++:

```
TAU_PROFILE_CALLSTACK( );
```

## Description

When TAU is configured with -PROFILECALLSTACK configuration option, and this call is invoked, a callpath trace is generated. A GUI for viewing this trace is included in TAU's utils/csUI directory. This option is deprecated.

## Example

C/C++ :

```
TAU_PROFILE_CALLSTACK( );
```

---

## Name

TAU\_TRACE\_RECVMSG -- Traces a receive operation

C/C++:

```
TAU_TRACE_RECVMSG(tag, source, length);
int tag;
int source;
int length;
```

Fortran:

```
TAU_TRACE_RECVMSG(tag, source, length);
integer tag;
integer source;
integer length;
```

## Description

TAU\_TRACE\_RECVMSG traces a receive operation where tag represents the type of the message received from the source process.

*NOTE:* When TAU is configured to use MPI (-mpiinc=<dir> -mpilib=<dir>), the TAU\_TRACE\_RECVMSG and TAU\_TRACE\_SENDMSG macros are not required. The wrapper interposition library in

```
$(TAU_MPI_LIBS)
```

uses these macros internally for logging messages.

## Example

C/C++:

```
if (pid == 0) {
    TAU_TRACE_SENDMSG(currCol, sender, ncols * sizeof(T));
    MPI_Send(vctr2, ncols * sizeof(T), MPI_BYTE, sender,
             currCol, MPI_COMM_WORLD);
} else {
    MPI_Recv(&ans, sizeof(T), MPI_BYTE, MPI_ANY_SOURCE,
             MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
    MPI_Get_count(&stat, MPI_BYTE, &recvcount);
    TAU_TRACE_RECVMSG(stat.MPI_TAG, stat.MPI_SOURCE, recvcount);
}
```

Fortran :

```
call TAU_TRACE_RECVMSG(tag, source, length)
call TAU_TRACE_SENDMSG(tag, destination, length)
```

## See Also

TAU\_TRACE\_SENDSMSG

---

## Name

TAU\_TRACE\_SENDSMSG -- Traces a receive operation

C/C++:

```
TAU_TRACE_SENDSMSG(tag, source, length);
int tag;
int source;
int length;
```

Fortran:

```
TAU_TRACE_SENDSMSG(tag, source, length);
integer tag;
integer source;
integer length;
```

## Description

TAU\_TRACE\_SENDSMSG traces an inter-process message communication when a tagged message is sent to a destination process.

*NOTE:* When TAU is configured to use MPI (-mpiinc=<dir> -mpilib=<dir>), the TAU\_TRACE\_SENDSMSG and TAU\_TRACE\_RECVMSG macros are not required. The wrapper interposition library in

```
$(TAU_MPI_LIBS)
```

uses these macros internally for logging messages.

## Example

C/C++:

```
if (pid == 0) {
    TAU_TRACE_SENDSMSG(currCol, sender, ncols * sizeof(T));
    MPI_Send(vctr2, ncols * sizeof(T), MPI_BYTE, sender,
             currCol, MPI_COMM_WORLD);
} else {
    MPI_Recv(&ans, sizeof(T), MPI_BYTE, MPI_ANY_SOURCE,
             MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
    MPI_Get_count(&stat, MPI_BYTE, &recvcount);
    TAU_TRACE_RECVMSG(stat.MPI_TAG, stat.MPI_SOURCE, recvcount);
}
```

Fortran :

```
call TAU_TRACE_RECVMSG(tag, source, length)
call TAU_TRACE_SENDSMSG(tag, destination, length)
```

## See Also

[TAU\\_TRACE\\_RECVMSG](#)

---

## Name

TAU\_PROFILE\_PARAM1L -- Creates a snapshot of the current application profile

C/C++:

```
TAU_PROFILE_PARAM1L(number, name);
long number;
char* name;
```

Fortran:

```
TAU_PROFILE_PARAM1L(name, number, length);
char* name;
integer number;
integer length;
```

## Description

Track the a given numeral parameter to a function and records each value as a seperate event. `number` is the parameter to be tracked. `name` is the name of this event.

## Example

C/C++:

```
int f1(int x)
{
    TAU_PROFILE("f1()", "", TAU_USER);
    TAU_PROFILE_PARAM1L((long) x, "x");
    ...
}
```

Fortran:

```
subroutine ITERATION(val)
    integer val
    integer profiler(2) / 0, 0 /
    save profiler

    call TAU_PROFILE_TIMER(profiler, 'INTERATION')
    call TAU_PROFILE_START(profiler)

    call TAU_PROFILE_PARAM1L('value', val, 4)

    ...
    call TAU_PROFILE_STOP(profiler)
    return
end
```

## See Also

TAU\_PROFILE\_TIMER\_DYNAMIC

---

## Name

TAU\_PROFILE\_SNAPSHOT -- Creates a snapshot of the current application profile

C/C++:

```
TAU_PROFILE_SNAPSHOT(name);
char* name;
```

Fortran:

```
TAU_PROFILE_SNAPSHOT(name, length);
char* name;
integer length;
```

## Description

TAU\_PROFILE\_SNAPSHOT writes a snapshot profile representing the program's execution up to this point. These files are written in the system as snapshot.[node].[context].[thread] format. They can be merged by appending one to another. Uploading a snapshot to a PerfDMF database or packing them into a PPK file will condense them to a single profile (the last one).

## Examples

C/C++:

```
TAU_PROFILE_SNAPSHOT(name);
```

Fortran:

```
TAU_PROFILE_SNAPSHOT(name, length);
```

Python:

```
import pytau;
pytau.snapshot("name")
```

## See Also

TAU\_PROFILE\_SNAPSHOT\_1L

---

## Name

TAU\_PROFILE\_SNAPSHOT\_1L -- Creates a snapshot of the current application profile

C/C++:

```
TAU_PROFILE_SNAPSHOT_1L(name, number);
char* name;
int number;
```

Fortran:

```
TAU_PROFILE_SNAPSHOT_1L(name, number, length);
char* name;
integer number;
integer length;
```

## Description

Calls TAU\_PROFILE\_SNAPSHOT giving it the as a name the name with a number appended.

## See Also

TAU\_PROFILE\_SNAPSHOT

---

## Name

TAU\_PROFILER\_CREATE -- Creates a profiler object referenced as a standard pointer

C/C++:

```
TAU_PROFILER_CREATE(timer, function_name, type, group);
Timer timer;
char* or string& function_name;
char* or string& type;
taugroup_t group;
```

## description

TAU\_PROFILER\_CREATE creates a timer the that can be controlled by the Timer pointer object.

The TAU\_PROFILER\_\* API is intended for applications to easily layer their legacy timing measurements APIs on top of TAU. Unlike other TAU API calls (TAU\_PROFILE\_TIMER) that are statically expanded in the source code, these calls allocate TAU entities on the heap. So the pointer to the TAU timer may be used as a handle to access the TAU performance data.

## example

>C/C++:

```
void *ptr;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);

TAU_PROFILER_START(ptr);
foo(2);
TAU_PROFILER_STOP(ptr);
```

>Python:

```
import pytau
ptr = pytau.profileTimer("foo")

pytau.start(ptr)
foo(2)
pytau.stop(ptr)
```

## See Also

TAU_PROFILER_START	TAU_PROFILER_STOP	TAU_PROFILER_GET_CALLS
TAU_PROFILER_GET_CHILD_CALLS	TAU_PROFILER_GET_INCLUSIVE_VALUES	
TAU_PROFILER_GET_EXCLUSIVE_VALUES	TAU_PROFILER_GET_COUNTER_INFO	

---

## Name

TAU\_CREATE\_TASK -- Creates a task id.

C/C++:

```
TAU_CREATE_TASK(taskid);
Integer taskid;
```

## description

TAU\_CREATE\_TASK creates a task with id 'taskid' this task is an independent event stream for which Profiler objects can be started and stop on. TAU will increment the taskids as needed and write out profiles and traces from the task as if they were thread.

## example

>C/C++:

```
void *ptr;
int taskid;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);
TAU_CREATE_TASK(taskid);
TAU_PROFILER_START_TASK(ptr,taskid);
foo(2);
TAU_PROFILER_STOP_TASK(ptr,taskid);
```

## See Also

TAU_PROFILER_START_TASK	TAU_PROFILER_STOP_TASK
TAU_PROFILER_GET_CALLS_TASK	TAU_PROFILER_GET_CHILD_CALLS_TASK
TAU_PROFILER_GET_INCLUSIVE_VALUES_TASK	
TAU_PROFILER_GET_EXCLUSIVE_VALUES_TASK	
TAU_PROFILER_GET_COUNTER_INFO_TASK	

---

## Name

TAU\_PROFILER\_START -- starts a profiler object created by TAU\_PROFILER\_CREATE

C/C++:

```
TAU_PROFILER_START(timer);
Timer timer;
```

## description

TAU\_PROFILER\_START starts a profiler timer by passing the pointer created by the TAU\_PROFILER\_CREATE.

## example

>C/C++:

```
void *ptr;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);

TAU_PROFILER_START(ptr);
foo(2);
TAU_PROFILER_STOP(ptr);
```

>Python:

```
import pytau
ptr = pytau.profileTimer("foo")

pytau.start(ptr)
foo(2)
pytau.stop(ptr)
```

## See Also

TAU_PROFILER_CREATE	TAU_PROFILER_STOP	TAU_PROFILER_GET_CALLS
TAU_PROFILER_GET_CHILD_CALLS	TAU_PROFILER_GET_INCLUSIVE_VALUES	
TAU_PROFILER_GET_EXCLUSIVE_VALUES	TAU_PROFILER_GET_COUNTER_INFO	

---

## Name

TAU\_PROFILER\_START\_TASK -- Starts a profiler object created by TAU\_PROFILER\_CREATE on a given task.

C/C++:

```
TAU_PROFILER_START_TASK(timer);
Timer timer;
```

## description

TAU\_PROFILER\_START\_TASK starts a profiler timer on a task by passing the pointer created by the TAU\_PROFILER\_CREATE and a task created by TAU\_CREATE\_TASK on a given task.

## example

>C/C++:

```
void *ptr;
int taskid;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);
TAU_CREATE_TASK(taskid);
TAU_PROFILER_START_TASK(ptr,taskid);
foo(2);
TAU_PROFILER_STOP_TASK(ptr,taskid);
```

## See Also

TAU_PROFILER_CREATE	TAU_PROFILER_STOP	TAU_PROFILER_GET_CALLS
TAU_PROFILER_GET_CHILD_CALLS	TAU_PROFILER_GET_INCLUSIVE_VALUES	
TAU_PROFILER_GET_EXCLUSIVE_VALUES	TAU_PROFILER_GET_COUNTER_INFO	

---

## Name

TAU\_PROFILER\_STOP -- stops a profiler object created by TAU\_PROFILER\_CREATE

C/C++:

```
TAU_PROFILER_STOP(timer);
Timer timer;
```

## description

TAU\_PROFILER\_STOP stops a profiler timer by passing the pointer created by the TAU\_PROFILER\_CREATE.

## example

>C/C++:

```
void *ptr;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);

TAU_PROFILER_START(ptr);
foo(2);
TAU_PROFILER_STOP(ptr);
```

>Python:

```
import pytau
ptr = pytau.profileTimer("foo")

pytau.start(ptr)
foo(2)
pytau.stop(ptr)
```

## See Also

TAU\_PROFILER\_CREATE      TAU\_PROFILER\_START      TAU\_PROFILER\_GET\_CALLS  
TAU\_PROFILER\_GET\_CHILD\_CALLS      TAU\_PROFILER\_GET\_INCLUSIVE\_VALUES  
TAU\_PROFILER\_GET\_EXCLUSIVE\_VALUES TAU\_PROFILER\_GET\_COUNTER\_INFO

---

## Name

TAU\_PROFILER\_STOP\_TASK -- Stops a profiler object on a task

C/C++:

```
TAU_PROFILER_STOP_TASK(timer);
Timer timer;
```

## description

TAU\_PROFILER\_STOP\_TASKSTOPS a profiler timer on a task by passing the pointer created by the TAU\_PROFILER\_CREATE and a task created by TAU\_CREATE\_TASK.

## example

>C/C++:

```
void *ptr;
int taskid;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);
TAU_CREATE_TASK(taskid);
TAU_PROFILER_START_TASK(ptr,taskid);
foo(2);
TAU_PROFILER_STOP_TASK(ptr,taskid);
```

## See Also

TAU_PROFILER_CREATE	TAU_PROFILER_STOP	TAU_PROFILER_GET_CALLS
TAU_PROFILER_GET_CHILD_CALLS	TAU_PROFILER_GET_INCLUSIVE_VALUES	
TAU_PROFILER_GET_EXCLUSIVE_VALUES	TAU_PROFILER_GET_COUNTER_INFO	

---

## Name

TAU\_PROFILER\_GET\_CALLS -- Gets the number of times this timer, created by TAU\_PROFILER\_CREATE, is started.

C/C++:

```
TAU_PROFILER_GET_CALLS(timer, calls);
Timer timer;
long& calls;
```

## description

TAU\_PROFILER\_GET\_CALLS returns the number of times this timer is started (ie. The number of times the section of code being profiled was executed).

## example

>C/C++:

```
void *ptr;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);

TAU_PROFILER_START(ptr);
foo(2);
long calls;
TAU_PROFILER_GET_CALLS(ptr, &calls);
```

## See Also

TAU_PROFILER_CREATE	TAU_PROFILER_START	TAU_PROFILER_STOP
TAU_PROFILER_GET_CHILD_CALLS	TAU_PROFILER_GET_INCLUSIVE_VALUES	
TAU_PROFILER_GET_EXCLUSIVE_VALUES	TAU_PROFILER_GET_COUNTER_INFO	

---

## Name

TAU\_PROFILER\_GET\_CALLS\_TASK -- Gets the number of times this timer, created by TAU\_PROFILER\_CREATE, is started on a given task.

C/C++:

```
TAU_PROFILER_GET_CALLS_TASK(timer, calls, taskid);
Timer timer;
long& calls;
int taskid;
```

## description

TAU\_PROFILER\_GET\_CALLS\_TASK returns the number of times this timer is started (ie. The number of times the section of code being profiled was executed) on a given task.

## example

>C/C++:

```
void *ptr;
int taskid;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);
TAU_CREATE_TASK(taskid);
TAU_PROFILER_START_TASK(ptr, taskid);
foo(2);
long calls;
TAU_PROFILER_GET_CALLS_TASK(ptr, &calls, taskid);
```

## See Also

TAU_CREATE_TASK	TAU_PROFILER_START_TASK	TAU_PROFILER_STOP_TASK
TAU_PROFILER_GET_CHILD_CALLS_TASK		
TAU_PROFILER_GET_INCLUSIVE_VALUES_TASK		
TAU_PROFILER_GET_EXCLUSIVE_VALUES_TASK		
TAU_PROFILER_GET_COUNTER_INFO_TASK		

---

## Name

TAU\_PROFILER\_GET\_CHILD\_CALLS -- Gets the number of calls made while this timer was running

C/C++:

```
TAU_PROFILER_GET_CHILD_CALLS(timer, calls);
Timer timer;
long& calls;
```

## description

TAU\_PROFILER\_GET\_CHILD\_CALLS Gets the number of timers started while `timer` was running.  
This is non-recursive, only timers started directly count.

## example

>C/C++:

```
void *ptr;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);

TAU_PROFILER_START(ptr);
foo(2);
TAU_PROFILER_STOP(ptr);

long calls;
TAU_PROFILER_GET_CHILD_CALLS(ptr, &calls);
```

## See Also

TAU_PROFILER_CREATE	TAU_PROFILER_START	TAU_PROFILER_STOP
TAU_PROFILER_GET_CALLS	TAU_PROFILER_GET_INCLUSIVE_VALUES	
TAU_PROFILER_GET_EXCLUSIVE_VALUES	TAU_PROFILER_GET_COUNTER_INFO	

---

## Name

TAU\_PROFILER\_GET\_CHILD\_CALLS\_TASK -- Gets the number of child call for this timer, created by TAU\_PROFILER\_CREATE, is started on a task.

C/C++:

```
TAU_PROFILER_GET_CHILD_CALLS_TASK(timer, child_calls, taskid);
Timer timer;
long& child_calls;
int taskid;
```

## description

TAU\_PROFILER\_GET\_CHILD\_CALLS\_TASK returns the number of times this timer is started (ie. The number of times the section of code being profiled was executed).

## example

>C/C++:

```
void *ptr;
int taskid;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);
TAU_CREATE_TASK(taskid);
TAU_PROFILER_START_TASK(ptr, taskid);
foo(2);
long child_calls;
TAU_PROFILER_GET_CHILD_CALLS_TASK(ptr, &child_calls, taskid);
```

## See Also

TAU\_CREATE\_TASK      TAU\_PROFILER\_START\_TASK      TAU\_PROFILER\_STOP\_TASK  
TAU\_PROFILER\_GET\_CALLS\_TASK      TAU\_PROFILER\_GET\_INCLUSIVE\_VALUES\_TASK  
TAU\_PROFILER\_GET\_EXCLUSIVE\_VALUES\_TASK  
TAU\_PROFILER\_GET\_COUNTER\_INFO\_TASK

---

## Name

TAU\_PROFILER\_GET\_INCLUSIVE\_VALUES -- Returns the inclusive amount of a metric spend by this timer.

C/C++:

```
TAU_PROFILER_GET_INCLUSIVE_VALUES(timer, incl);
Timer timer;
double& incl;
```

## description

TAU\_PROFILER\_GET\_INCLUSIVE\_VALUES Returns the inclusive amount of a metric spend while this timer was running (and any subsequent timers called from this timer.)

## example

>C/C++:

```
void *ptr;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);

TAU_PROFILER_START(ptr);
foo(2);
TAU_PROFILER_STOP(ptr);

double incl[TAU_MAX_COUNTERS];
TAU_PROFILER_GET_INCLUSIVE_VALUES(ptr, &incl);
```

## See Also

TAU_PROFILER_CREATE	TAU_PROFILER_START	TAU_PROFILER_STOP
TAU_PROFILER_GET_CALLS		TAU_PROFILER_GET_CHILD_CALLS
TAU_PROFILER_GET_EXCLUSIVE_VALUES	TAU_PROFILER_GET_COUNTER_INFO	

---

## Name

TAU\_PROFILER\_GET\_INCLUSIVE\_VALUES\_TASK -- Returns the inclusive amount of a metric spend by this timer on a given task.

C/C++:

```
TAU_PROFILER_GET_INCLUSIVE_VALUES_TASK(timer, incl, taskid);
Timer timer;
double& incl;
int taskid;
```

## description

TAU\_PROFILER\_GET\_INCLUSIVE\_VALUES\_TASK Returns the inclusive amount of a metric spend while this timer was running (and any subsequent timers called from this timer) on a given task.

## example

>C/C++:

```
void *ptr;
int taskid;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);
TAU_CREATE_TASK(taskid);
TAU_PROFILER_START(ptr);
foo(2);
TAU_PROFILER_STOP(ptr);

double incl[TAU_MAX_COUNTERS];
TAU_PROFILER_GET_INCLUSIVE_VALUES_TASK(ptr, &incl, taskid);
```

## See Also

TAU_CREATE_TASK	TAU_PROFILER_START_TASK	TAU_PROFILER_STOP_TASK
TAU_PROFILER_GET_CALLS_TASK		TAU_PROFILER_GET_CHILD_CALLS_TASK
TAU_PROFILER_GET_EXCLUSIVE_VALUES_TASK		
TAU_PROFILER_GET_COUNTER_INFO_TASK		

---

## Name

TAU\_PROFILER\_GET\_EXCLUSIVE\_VALUES -- Returns the exclusive amount of a metric spend by this timer.

C/C++:

```
TAU_PROFILER_GET_EXCLUSIVE_VALUES(timer, excl);
Timer timer;
double& excl;
```

## description

TAU\_PROFILER\_GET\_EXCLUSIVE\_VALUES Returns the exclusive amount of the metric spend while this timer was running (and while no other subsequent timers was running.)

## example

>C/C++:

```
void *ptr;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);

TAU_PROFILER_START(ptr);
foo(2);
TAU_PROFILER_STOP(ptr);

double excl[TAU_MAX_COUNTERS];
TAU_PROFILER_GET_EXCLUSIVE_VALUES(ptr, &excl);
```

## See Also

TAU_PROFILER_CREATE	TAU_PROFILER_START	TAU_PROFILER_STOP
TAU_PROFILER_GET_CALLS		TAU_PROFILER_GET_CHILD_CALLS
TAU_PROFILER_GET_INCLUSIVE_VALUES	TAU_PROFILER_GET_COUNTER_INFO	

---

## Name

TAU\_PROFILER\_GET\_EXCLUSIVE\_VALUES\_TASK -- Returns the exclusive amount of a metric spend by this timer on a given task.

C/C++:

```
TAU_PROFILER_GET_EXCLUSIVE_VALUES_TASK(timer, excl, taskid);
Timer timer;
double& excl;
int taskid;
```

## description

TAU\_PROFILER\_GET\_EXCLUSIVE\_VALUES\_TASK Returns the exclusive amount of the metric spend while this timer was running (and while no other subsequent timers was running) on a given task.

## example

>C/C++:

```
void *ptr;
int taskid;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);
TAU_CREATE_TASK(taskid);
TAU_PROFILER_START(ptr);
foo(2);
TAU_PROFILER_STOP(ptr);

double excl[TAU_MAX_COUNTERS];
TAU_PROFILER_GET_EXCLUSIVE_VALUES_TASK(ptr, &excl, taskid);
```

## See Also

TAU_PROFILER_CREATE	TAU_PROFILER_START	TAU_PROFILER_STOP
TAU_PROFILER_GET_CALLS		TAU_PROFILER_GET_CHILD_CALLS
TAU_PROFILER_GET_INCLUSIVE_VALUES	TAU_PROFILER_GET_COUNTER_INFO	

---

## Name

TAU\_PROFILER\_GET\_COUNTER\_INFO -- Returns information about all the timers created.

C/C++:

```
TAU_PROFILER_GET_COUNTER_INFO(counters, num_counters);
const char * counters;
int &num_counters;
```

## description

TAU\_PROFILER\_GET\_COUNTER\_INFO Gets the number of counters created and an array of the counters containing information about the counters.

## example

>C/C++:

```
void *ptr;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);

TAU_PROFILER_START(ptr);
foo(2);
TAU_PROFILER_STOP(ptr);

const char **counters;
int numcounters;

TAU_PROFILER_GET_COUNTER_INFO(&counters, &numcounters);
printf("numcounters = %d\n", numcounters);
for (j = 0; j < numcounters ; j++)
{
    printf(">>>");
    printf("counter [%d] = %s\n", j, counters[j]);
}
```

## See Also

TAU_PROFILER_CREATE	TAU_PROFILER_START	TAU_PROFILER_STOP
TAU_PROFILER_GET_CALLS	TAU_PROFILER_GET_CHILD_CALLS	
TAU_PROFILER_GET_INCLUSIVE_VALUES	TAU_PROFILER_GET_EXCLUSIVE_VALUES	

---

## Name

TAU\_PROFILER\_GET\_COUNTER\_INFO\_TASK -- Returns information about all the timers created on a task.

C/C++:

```
TAU_PROFILER_GET_COUNTER_INFO_TASK(counters, num_counters, taskid);
const char * counters;
int &num_counters;
int taskid;
```

## description

TAU\_PROFILER\_GET\_COUNTER\_INFO\_TASK Gets the number of counters created and an array of the counters containing information about the counters on a given task.

## example

>C/C++:

```
void *ptr;
int taskid;
TAU_PROFILER_CREATE(ptr, "foo", "", TAU_USER);
TAU_CREATE_TASK(taskid);
TAU_PROFILER_START_TASK(ptr, taskid);
foo(2);
TAU_PROFILER_STOP_TASK(ptr, taskid);

const char **counters;
int numcounters;

TAU_PROFILER_GET_COUNTER_INFO_TASK(&counters, &numcounters, taskid);
printf("numcounters = %d\n", numcounters);
for (j = 0; j < numcounters ; j++)
{
    printf(">>>");
    printf("counter [%d] = %s\n", j, counters[j]);
}
```

## See Also

TAU\_CREATE\_TASK      TAU\_PROFILER\_START\_TASK      TAU\_PROFILER\_STOP\_TASK  
TAU\_PROFILER\_GET\_CALLS\_TASK      TAU\_PROFILER\_GET\_CHILD\_CALLS\_TASK  
TAU\_PROFILER\_GET\_INCLUSIVE\_VALUES\_TASK  
TAU\_PROFILER\_GET\_EXCLUSIVE\_VALUES\_TASK

---

## Name

TAU\_QUERY\_DECLARE\_EVENT -- Returns a event handle.

C/C++:

```
TAU_QUERY_DECLARE_EVENT(event);
void * event;
```

## description

TAU\_QUERY\_DECLARE\_EVENT Creates a event handle for querying TAU events.

## example

>C/C++:

```
char[100] str;
TAU_QUERY_DECLARE_EVENT(event);
TAU_QUERY_GET_CURRENT_EVENT(event);
TAU_QUERY_GET_EVENT_NAME(event, str);

printf("current event is: %d.\n", str);
```

## See Also

TAU\_QUERY\_DECLARE\_EVENT  
TAU\_QUERY\_GET\_EVENT\_NAME

TAU\_QUERY\_GET\_CURRENT\_EVENT  
TAU\_QUERY\_GET\_PARENT\_EVENT

---

## Name

TAU\_QUERY\_GET\_CURRENT\_EVENT -- set event to be the current TAU event.

C/C++:

```
TAU_QUERY_GET_CURRENT_EVENT(event);
void * event;
```

## description

TAU\_QUERY\_GET\_CURRENT\_EVENT Set event to be the current TAU event in the context in which this call is made.

## example

>C/C++:

```
char[100] str;
TAU_QUERY_DECLARE_EVENT(event);
TAU_QUERY_GET_CURRENT_EVENT(event);
TAU_QUERY_GET_EVENT_NAME(event, str);

printf("current event is: %d.\n", str);
```

## See Also

TAU_QUERY_DECLARE_EVENT	TAU_QUERY_GET_CURRENT_EVENT
TAU_QUERY_GET_EVENT_NAME	TAU_QUERY_GET_PARENT_EVENT

---

## Name

TAU\_QUERY\_GET\_EVENT\_NAME -- Gets the name of a given event.

C/C++:

```
TAU_QUERY_GET_EVENT_NAME(event, str);
void * event;
char * str;
```

## description

TAU\_QUERY\_GET\_EVENT\_NAME Set str to be the event name to the given event name.

## example

>C/C++:

```
char[100] str;
TAU_QUERY_DECLARE_EVENT(event);
TAU_QUERY_GET_CURRENT_EVENT(event);
TAU_QUERY_GET_EVENT_NAME(event, str);

printf("current event is: %d.\n", str);
```

## See Also

TAU\_QUERY\_DECLARE\_EVENT                            TAU\_QUERY\_GET\_CURRENT\_EVENT  
TAU\_QUERY\_GET\_EVENT\_NAME TAU\_QUERY\_GET\_PARENT\_EVENT

---

## Name

TAU\_QUERY\_GET\_PARENT\_EVENT -- gets the parent of the current event.

C/C++:

```
TAU_QUERY_GET_PARENT_EVENT(event);
void * event;
```

## description

TAU\_QUERY\_GET\_PARENT\_EVENT Set event to be the parent event to the current event.

## example

>C/C++:

```
char[100] str;
TAU_QUERY_DECLARE_EVENT(event);
TAU_QUERY_GET_PARENT_EVENT(event);
TAU_QUERY_GET_EVENT_NAME(event, str);

printf("parent event is: %d.\n", str);
```

## See Also

TAU\_QUERY\_DECLARE\_EVENT  
TAU\_QUERY\_GET\_EVENT\_NAME

TAU\_QUERY\_GET\_CURRENT\_EVENT  
TAU\_QUERY\_GET\_PARENT\_EVENT

---

# TAU Mapping API

## Introduction

TAU allows the user to map performance data of entities from one layer to another in multi-layered software. Mapping is used in profiling (and tracing) both synchronous and asynchronous models of computation.

For mapping, the following macros are used. First locate and identify the higher-level statement using the TAU\_MAPPING macro. Then, associate a function identifier with it using the TAU\_MAPPING\_OBJECT. Associate the high level statement to a FunctionInfo object that will be visible to lower level code, using TAU\_MAPPING\_LINK, and then profile entire blocks using TAU\_MAPPING\_PROFILE. Independent sets of statements can be profiled using TAU\_MAPPING\_PROFILE\_TIMER, TAU\_MAPPING\_PROFILE\_START, and TAU\_MAPPING\_PROFILE\_STOP macros using the FunctionInfo object.

The TAU examples/mapping directory has two examples (embedded and external) that illustrate the use of this mapping API for generating object-oriented profiles.

---

## Name

TAU\_MAPPING -- Encapsulates a C++ statement for profiling

C/C++:

```
TAU_MAPPING(statement, key);
statement statement;
TauGroup_t key;
```

## Description

TAU\_MAPPING is used to encapsulate a C++ statement as a timer. A timer will be made, named by the statement, and will profile the statement. The key given can be used with TAU\_MAPPING\_LINK to retrieve the timer.

## Example

C/C++:

```
int main(int argc, char **argv) {
    Array <2> A(N, N), B(N, N), C(N,N), D(N, N);
    // Original statement:
    // A = B + C + D;
    // Instrumented statement:
    TAU_MAPPING(A = B + C + D; , TAU_USER);
    ...
}
```

## See Also

TAU\_MAPPING\_CREATE, TAU\_MAPPING\_LINK

---

## Name

TAU\_MAPPING\_CREATE -- Creates a mapping

C/C++:

```
TAU_MAPPING_CREATE(name, type, groupname, key, tid);
char *name;
char *type;
char *groupname;
unsigned long key;
int tid;
```

## Description

TAU\_MAPPING\_CREATE creates a mapping and associates it with the key that is specified. Later, this key may be used to retrieve the FunctionInfo object associated with this key for timing purposes. The thread identifier is specified in the tid parameter.

## Example

C/C++:

```
class MyClass {
public:
    MyClass() {
        TAU_MAPPING_LINK(runtimer, TAU_USER);
    }
    ~MyClass() {}

    void Run(void) {
        TAU_MAPPING_PROFILE(runtimer); // For one object
        TAU_PROFILE("MyClass::Run()", " void (void)", TAU_USER1);

        cout << "Sleeping for 2 secs..." << endl;
        sleep(2);
    }
private:
    TAU_MAPPING_OBJECT(runtimer) // EMBEDDED ASSOCIATION
};

int main(int argc, char **argv) {
    TAU_PROFILE_INIT(argc, argv);
    TAU_PROFILE("main()", "int (int, char **)", TAU_DEFAULT);
    MyClass x, y, z;
    TAU_MAPPING_CREATE("MyClass::Run() for object a", " ", TAU_USER,
                       "TAU_USER", 0);
    MyClass a;
    TAU_PROFILE_SET_NODE(0);
    cout << "Inside main" << endl;

    a.Run();
    x.Run();
    y.Run();
}
```

## See Also

TAU\_MAPPING\_LINK, TAU\_MAPPING\_OBJECT, TAU\_MAPPING\_PROFILE

---

## Name

TAU\_MAPPING\_LINK -- Creates a mapping link

C/C++:

```
TAU_MAPPING_LINK(FuncIdVar, Key);
FunctionInfo FuncIdVar;
unsigned long Key;
```

## Description

TAU\_MAPPING\_LINK creates a link between the object defined in TAU\_MAPPING\_OBJECT (that identifies a statement) and the actual higher-level statement that is mapped with TAU\_MAPPING. The Key argument represents a profile group to which the statement belongs, as specified in the TAU\_MAPPING macro argument. For the example of array statements, this link should be created in the constructor of the class that represents the expression. TAU\_MAPPING\_LINK should be executed before any measurement takes place. It assigns the identifier of the statement to the object to which FuncIdVar refers. For example

## Example

C/C++:

```
class MyClass {
public:
    MyClass() { }
    ~MyClass() { }

    void Run(void) {
        TAU_MAPPING_OBJECT(runtimer)
        TAU_MAPPING_LINK(runtimer, (unsigned long) this);
        TAU_MAPPING_PROFILE(runtimer); // For one object
        TAU_PROFILE("MyClass::Run()", " void (void)", TAU_USER1);

        /* ... */
    }
};

int main(int argc, char **argv) {
    TAU_PROFILE_INIT(argc, argv);
    TAU_PROFILE("main()", "int (int, char **)", TAU_DEFAULT);
    MyClass x, y, z;
    MyClass a;
    TAU_MAPPING_CREATE("MyClass::Run() for object a", " " ,
                       (TauGroup_t) &a, "TAU_USER", 0);
    TAU_MAPPING_CREATE("MyClass::Run() for object x", " " ,
                       (TauGroup_t) &x, "TAU_USER", 0);
    TAU_PROFILE_SET_NODE(0);
    cout << "Inside main" << endl;

    a.Run();
    x.Run();
    y.Run();
}
```

## See Also

TAU\_MAPPING\_CREATE, TAU\_MAPPING\_OBJECT, TAU\_MAPPING\_PROFILE

---

## Name

TAU\_MAPPING\_OBJECT -- Declares a mapping object

C/C++:

```
TAU_MAPPING_OBJECT(FuncIdVar);
FunctionInfo FuncIdVar;
```

## Description

To create storage for an identifier associated with a higher level statement that is mapped using TAU\_MAPPING, we use the TAU\_MAPPING\_OBJECT macro. For example, in the TAU\_MAPPING example, the array expressions are created into objects of a class ExpressionKernel, and each statement is an object that is an instance of this class. To embed the identity of the statement we store the mapping object in a data field in this class. This is shown below:

## Example

C/C++:

```
template<class LHS, class Op, class RHS, class EvalTag>
class ExpressionKernel : public Pooma::Iterate_t {
public:
    typedef ExpressionKernel<LHS,Op,RHS,EvalTag> This_t;
    //
    // Construct from an Expr.
    // Build the kernel that will evaluate the expression on the
    // given domain.
    // Acquire locks on the data referred to by the expression.
    //
    ExpressionKernel(const LHS&, const Op&, const RHS&,
                     Pooma::Scheduler_t&);

    virtual ~ExpressionKernel();

    // Do the loop.
    virtual void run();

private:
    // The expression we will evaluate.
    LHS lhs_m;
    Op op_m;
    RHS rhs_m;
    TAU_MAPPING_OBJECT(TauMapFI)
};
```

## See Also

TAU\_MAPPING\_CREATE, TAU\_MAPPING\_LINK, TAU\_MAPPING\_PROFILE

---

## Name

TAU\_MAPPING\_PROFILE -- Profiles a block based on a mapping

C/C++:

```
TAU_MAPPING_PROFILE(FuncIdVar);  
FunctionInfo *FuncIdVar;
```

## Description

The TAU\_MAPPING\_PROFILE macro measures the time and attributes it to the statement mapped in TAU\_MAPPING macro. It takes as its argument the identifier of the higher level statement that is stored using TAU\_MAPPING\_OBJECT and linked to the statement using TAU\_MAPPING\_LINK macros. TAU\_MAPPING\_PROFILE measures the time spent in the entire block in which it is invoked. For example, if the time spent in the run method of the class does work that must be associated with the higher-level array expression, then, we can instrument it as follows:

## Example

C/C++ :

```
// Evaluate the kernel  
// Just tell an InlineEvaluator to do it.  
  
template<class LHS, class Op, class RHS, class EvalTag>  
void  
ExpressionKernel<LHS,Op,RHS,EvalTag>::run() {  
    TAU_MAPPING_PROFILE(TauMapFI)  
  
    // Just evaluate the expression.  
    KernelEvaluator<EvalTag>().evaluate(lhs_m,op_m,rhs_m);  
    // we could release the locks here or in dtor  
}
```

## See Also

TAU\_MAPPING\_CREATE, TAU\_MAPPING\_LINK, TAU\_MAPPING\_OBJECT

---

## Name

TAU\_MAPPING\_PROFILE\_START -- Starts a mapping timer

C/C++:

```
TAU_MAPPING_PROFILE_START(timer, tid);
Profiler timer;
int tid;
```

## Description

TAU\_MAPPING\_PROFILE\_START starts the timer that is created using TAU\_MAPPING\_PROFILE\_TIMER. This will measure the elapsed time in groups of statements, instead of the entire block. A corresponding stop statement stops the timer as described next. The thread identifier is specified in the tid parameter.

## Example

C/C++:

```
template<class LHS, class Op, class RHS, class EvalTag>
void
ExpressionKernel<LHS, Op, RHS, EvalTag>::run() {
    TAU_MAPPING_PROFILE_TIMER(timer, TauMapFI);
    printf("ExpressionKernel::run() this = 4854\n", this);
    // Just evaluate the expression.

    TAU_MAPPING_PROFILE_START(timer);
    KernelEvaluator<EvalTag>().evaluate(lhs_m, op_m, rhs_m);
    TAU_MAPPING_PROFILE_STOP();
    // we could release the locks here instead of in the dtor.
}
```

## See Also

TAU\_MAPPING\_PROFILE\_TIMER, TAU\_MAPPING\_PROFILE\_STOP

---

## Name

TAU\_MAPPING\_PROFILE\_STOP -- Stops a mapping timer

C/C++:

```
TAU_MAPPING_PROFILE_STOP(timer, tid);
Profiler timer;
int tid;
```

## Description

TAU\_MAPPING\_PROFILE\_STOP stops the timer that is created using TAU\_MAPPING\_PROFILE\_TIMER. This will measure the elapsed time in groups of statements, instead of the entire block. A corresponding stop statement stops the timer as described next. The thread identifier is specified in the tid parameter.

## Example

C/C++:

```
template<class LHS, class Op, class RHS, class EvalTag>
void
ExpressionKernel<LHS, Op, RHS, EvalTag>::run() {
    TAU_MAPPING_PROFILE_TIMER(timer, TauMapFI);
    printf("ExpressionKernel::run() this = 4854\n", this);
    // Just evaluate the expression.

    TAU_MAPPING_PROFILE_START(timer);
    KernelEvaluator<EvalTag>().evaluate(lhs_m, op_m, rhs_m);
    TAU_MAPPING_PROFILE_STOP();
    // we could release the locks here instead of in the dtor.
}
```

## See Also

TAU\_MAPPING\_PROFILE\_TIMER, TAU\_MAPPING\_PROFILE\_START

---

## Name

TAU\_MAPPING\_PROFILE\_TIMER -- Declares a mapping timer

C/C++:

```
TAU_MAPPING_PROFILE_TIMER(timer, FuncIdVar);
Profiler timer;
FunctionInfo *FuncIdVar;
```

## Description

TAU\_MAPPING\_PROFILE\_TIMER enables timing of individual statements, instead of complete blocks. It will attribute the time to a higher-level statement. The second argument is the identifier of the statement that is obtained after TAU\_MAPPING\_OBJECT and TAU\_MAPPING\_LINK have executed. The timer argument in this macro is any variable that is used subsequently to start and stop the timer.

## Example

C/C++:

```
template<class LHS, class Op, class RHS, class EvalTag>
void
ExpressionKernel<LHS,Op,RHS,EvalTag>::run() {
    TAU_MAPPING_PROFILE_TIMER(timer, TauMapFI);
    printf("ExpressionKernel::run() this = 4854\n", this);
    // Just evaluate the expression.

    TAU_MAPPING_PROFILE_START(timer);
    KernelEvaluator<EvalTag>().evaluate(lhs_m, op_m, rhs_m);
    TAU_MAPPING_PROFILE_STOP();
    // we could release the locks here instead of in the dtor.
}
```

## See Also

TAU\_MAPPING\_LINK,    TAU\_MAPPING\_OBJECT,    TAU\_MAPPING\_PROFILE\_START,  
TAU\_MAPPING\_PROFILE\_STOP

---

# Appendix A. Environment Variables

**Table A.1. TAU Environment Variables**

VARIABLE NAME	DESCRIPTION
TAU_PROFILE	Set to 1 to have TAU profile your code
TAU_TRACE	Set to 1 to have TAU trace your code
TAU_METRICS	Colon delimited list of TAU/PAPI metrics to profile
PAPI_EVENT	Sets the hardware counter to use when TAU is configured with -PAPI. See Section 2.6, “Using Hardware Performance Counters”
PCL_EVENT	Sets the hardware counter to use when TAU is configured with -PCL. See Section 2.6, “Using Hardware Performance Counters”
PROFILEDIR	Selectively measure groups of routines and statements. Use with -profile command line option. See ???
TAU_CALLPATH	When set to 1 TAU will generate call-path data. Use with TAU_CALLPATH_DEPTH.
TAU_CALLPATH_DEPTH	Sets the depth of the callpath profiling. Use with TAU_CALLPATH environment variable.
TAU_TRACK_MESSAGE	Track MPI message statistics (profiling), messages lines (tracing).
TAU_COMM_MATRIX	Generate MPI communication matrix data.
TAU_COMPENSATE	Attempt to compensate for profiling overhead in profiles.
TAU_COMPENSATE_ITERATIONS	Set the number of iterations TAU uses to estimate the measurement overhead. A larger number of iteration will increases profiling precision (default 1000).
TAU_KEEP_TRACEFILES	Retains the intermediate trace files. Use with -TRACE TAU configuration option. See ???
TAU_MUSE_PACKAGE	Sets the MAGNET/MUSE package name. Use with the -muse TAU configuration option. See ???
TAU_THROTTLE	Enables the runtime throttling of events that are lightweight. See ???
TAU_THROTTLE_NUMCALLS	Set the maximum number of calls that will be profiled for any function when TAU_THROTTLE is enabled. See ???
TAU_THROTTLE_PERCALL	Set the minimum inclusive time (in milliseconds) a function has to have to be instrumented when TAU_THROTTLE is enabled. See ???
TAU_TRACEFILE	Specifies the name of Vampir trace file. Use with -TRACE TAU configuration option. See ???
TRACEDIR	Specifies the directory where trace file are to be stored. See ???

VARIABLE NAME	DESCRIPTION
TAU_VERBOSE	When set TAU will print out information about its configuration when running a instrumented application.
TAU_PROFILE_FORMAT	When set to <code>snapshot</code> TAU will generate condensed snapshot profiles (they merge together different metrics so there is only one file per node.) Instead of the default kind. When set to <code>merged</code> , TAU will pre-compute mean and std. dev. at the end of execution.
TAU_SYNCHRONIZE_CLOCKS	When set TAU will correct for any time discrepancies between nodes because of their CPU clock lag. This should produce more reliable trace data.
TAU_SAMPLING	<p>Default value is 0 (off). When TAU_SAMPLING is set, we collect additional profile or trace information (depending on whether TAU_PROFILE or TAU_TRACE is set respectively) via periodic sampling at runtime. Metrics collected and sampling period is controlled by TAU_EBS_SOURCE and TAU_EBS_PERIOD variables respectively. The TAU_EBS_UNWIND variable determines if callstack unwinding is enabled at each sample.</p> <p>For TAU_PROFILE, in addition to regular TAU instrumented profile output, samples will show up as additional events prefixed by [SAMPLE] for each unique function, file and source line number combination. These events are grouped under [INTERMEDIATE] event nodes for the instrumented TAU context where the samples occurred. In addition, if TAU_EBS_UNWIND is active, [UNWIND] event nodes may be generated for each discovered callstack entry found by the callstack unwinder.</p> <p>TAU_SAMPLING is dependent on the availability of BFD as determined by the <code>-bfd</code> configuration option when building TAU. Its ability to resolve sample addresses into function, file name and source line number information may be limited or missing if BFD is missing or is installed with limited functionality. If in doubt, please try building TAU with <code>-bfd=download</code>. Any one of function, file name and source line number may be missing. In the event all three are, the event is marked as "UNRESOLVED". The TAU_EBS_KEEP_UNRESOLVED_ADDR variable enables addresses to be retained for unresolved results.</p>
TAU_EBS_SOURCE	Default value is "itimer". This variable sets the metric that determines the period of sampling. If the value is "itimer" (default), it represents the number of microseconds between samples (as determined by TAU_EBS_PERIOD). If the value is a PAPI metric (eg. PAPI_FP_INS), then it represents

VARIABLE NAME	DESCRIPTION
	the number of counts of that metric between samples (eg. every 10,000 floating-point instructions if PAPI_FP_INS is used). For "itimer", the samples occur as a result of system timer interrupts while for PAPI they occur in response to PAPI counter overflow interrupts set to the value of the TAU_EBS_PERIOD.
TAU_EBS_PERIOD	Default value is 0 (off). This enables callstack unwinding for each sample using the callstack unwinder specified at TAU configuration time. As of this writing, only the libunwind tool is supported. Support for other callstack unwinders like Stack-walkerAPI will be included. The TAU_EBS_UNWIND_DEPTH variable is used to control how many times the TAU sampling framework will be allowed to unwind the callstack.
TAU_EBS_UNWIND	Default value is 0 (off). This enables callstack unwinding for each sample using the callstack unwinder specified at TAU configuration time. As of this writing, only the libunwind tool is supported. Support for other callstack unwinders like Stack-walkerAPI will be included. The TAU_EBS_UNWIND_DEPTH variable is used to control how many times the TAU sampling framework will be allowed to unwind the callstack.
TAU_EBS_UNWIND_DEPTH	Default value is 10. This controls how many layers of the callstack TAU should unwind before attaching the result to the appropriate TAU event context.
TAU_EBS_KEEP_UNRESOLVED_ADDR	Default value is 0 (off). When set, this variable allows sample addresses that fail to be resolved by BFD to be recorded as "UNRESOLVED <modulename> ADDR <addr>" instead of "UNSOLVED <modulename>". This provides nominally more information than the default scenario in light of missing BFD information.
TAU_TRACK_SIGNALS	Set this variables to 1 to capture callstack as metadata at point of failure.
TAU_SUMMARY	Set this variables to 1 to generate just min/max/stddev/mean statistics instead of per-node data. Use paraprof -dumpsummary and then pprof -f profile.Max/Min to see the data.